Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Implementação de uma Linguagem Concorrente com Tipos Comportamentais

Nuno Jorge Corvo Parreira (28074)

Lisboa
(Fevereiro de 2011)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Implementação de uma Linguagem Concorrente com Tipos Comportamentais

Nuno Jorge Corvo Parreira (28074)

Orientador: Prof. Doutor João Costa Seco

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa
(Fevereiro de 2011)

Aos meus pais

Agradecimentos

Em primeiro lugar gostava de agradecer ao meu orientador, professor João Seco, por todo o apoio que deu ao longo da realização desta dissertação, pelo incentivo e perseverança que transmitiu ao longo do último ano e meio. Gostava também de agradecer a todos os colegas e amigos que encontrei no meu percurso, desde os tempos do "Pimpão" até hoje, pelos bons momentos juntos.

Obrigado ainda aos meus alunos . . . pelos cabelos brancos que me arranjaram ☺

O meu agradecimento especial . . .

. . . à Paula e ao Miguel por estarem sempre prontos a ajudar e aos seus filhos António, João e Júlia pelas brincadeiras e momentos bem passados,

. . . ao Castim por me "aprender" a ultrapassar as adversidades,

. . . a todos os que facilitaram a minha integração em Lisboa,

. . . ao GAN e a todos os que fazem ou já fizeram parte deste grande clube,

. . . e, como não poderia deixar de ser, ao SPORTING, o meu vício e esconderijo nos momentos complicados!

Este trabalho não teria sido possível sem a ajuda dos meus pais e do meu avô, que me proporcionaram todas as condições para prosseguir a minha carreira académica longe de casa. Obrigado por todo o apoio nas horas difíceis e por estarem sempre lá quando foi preciso!

Este trabalho foi parcialmente suportado pelas bolsas PTDC/EIA-CCO/104583/2008 e de introdução à investigação do DI-FCT-UNL.

Resumo

A programação concorrente com base em memória partilhada é uma disciplina difícil. A possível interferência entre *threads* no acesso a zonas de memória partilhada pode causar comportamentos que comprometam o funcionamento esperado de um programa. Em geral, pretende-se que os programas concorrentes não tenham problemas do tipo *deadlocks* ou *race conditions*. Esse objectivo atinge-se controlando o acesso às zonas partilhadas através de mecanismos fornecidos a nível dos sistemas operativos, que são eficazes mas difíceis de usar sem cometer erros.

Uma forma de minimizar esse problema é usar linguagens com abstracções próprias para representar concorrência, e mais ainda se estas permitirem efectuar análise estática do código dos programas para detectar situações anómalas.

Este trabalho descreve a implementação de uma linguagem de programação orientada aos objectos, com suporte para concorrência, e respectivo algoritmo de tipificação, baseado em tipos espaciais/comportamentais, com operadores de tipo de composição sequencial, composição paralela, escolha, repetição e replicação. O sistema de tipos em que se baseia o algoritmo de tipificação apresentado aqui, garante a ausência de *race conditions* nos programas através da disciplina no acesso a recursos partilhados. O algoritmo de tipificação combina técnicas de verificação de tipos com inferência de tipos; obtém também a utilização dos identificadores livres por análise das expressões, e relaciona-a com os tipos declarados para os objectos através de uma relação de *subtyping*.

Palavras-chave: Linguagens de programação, concorrência, memória partilhada, verificação estática, tipos espaciais/comportamentais, inferência de tipos, *subtyping*.

Abstract

Concurrent programming based on shared memory is a hard discipline. The possible interference between threads upon accessing shared variables can cause unexpected changes that may compromise a program behavior. Usually, we want that concurrent programs with shared memory don't raise problems like deadlocks or race conditions. To tackle this problem, programmers control shared memory access by using operating system primitives to serialize the accesses. Those primitives may be effective but error prone.

One approach to solve that issue is to use language abstractions to represent concurrency and try to detect anomalous situations at compile-time by static analysis of the program's source code.

We describe an implementation of an object oriented programming language and a typing algorithm to discipline the access to shared resources and hence avoid race conditions. This language is based on spatial/behavioral types with sequential, parallel, choice, repetition and replication operators. We introduce a typing algorithm based on type inference to to obtain the actual usage of objects, and compare it with the declared object protocols.

Keywords: Programming languages, concurrency, shared memory, static checking, spatial/behavioral types, type inference, subtyping.

Conteúdo

Lista de Figuras

Listagens

1

Introdução

A intenção deste trabalho é implementar uma linguagem de programação que ajude os programadores na tarefa não trivial de criar programas concorrentes. Em geral, pretende-se que o comportamento de cada um dos fluxos de execução (*threads*) de um programa concorrente seja perceptível através da leitura do seu código, como se fosse executado isoladamente, sem interferências ou em conjunto com outros mas em que sejam claros os pontos de sincronização. O código das várias *threads* de um programa concorrente pode e, muitas vezes, deve ser escrito para que as *threads* interajam. No entanto, esta interacção deve ser desenhada e programada explicitamente usando mecanismos próprios tais como *locks* (mecanismo de baixo nível para assegurar exclusão mútua), monitores (abstracções de programação) [Hoa78, Hoa74], ou implicitamente como no caso das transacções em memória [HMPJH05]. As várias *threads* de um programa não devem interferir umas com as outras de forma inesperada e desta maneira produzir comportamentos imprevisíveis.

O tipo de interferências mais comum e que se propõe tratar neste trabalho, é causado pela partilha não controlada de zonas de memória. Para evitar que dessas interferências resultem comportamentos inesperados dos programas (as chamadas *race conditions*), usam-se mecanismos de controlo durante a execução dos mesmos [GBB+06]. Esses mecanismos disciplinam o acesso aos recursos partilhados. Em programas com memória partilhada existem três abordagens possíveis para resolver este tipo de problemas: não partilhar memória entre *threads*, tornar as variáveis imutáveis ou utilizar mecanismos de sincronização. A sincronização é naturalmente suportada de forma directa pelos sistemas de operação e está presente em muitas linguagens de programação, de várias maneiras. Entre as técnicas mais usuais destacam-se os semáforos [Dij65], os monitores [Hoa74, Hoa78, Han93] ou as primitivas atómicas de *compare*

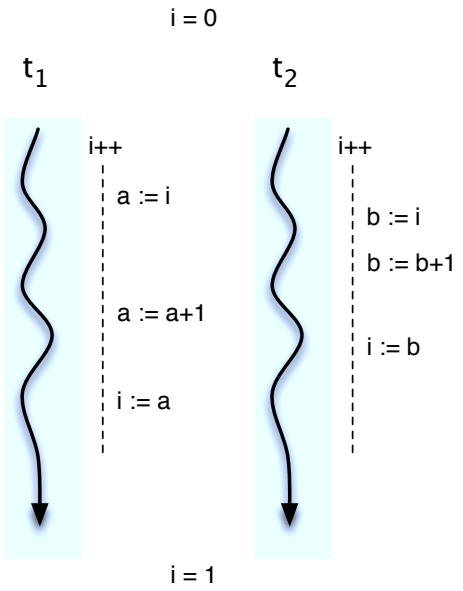

Figura 1.1: Exemplo de uma *race condition*

and swap, usadas nos chamados *lock-free algorithms* [MS96] e mecanismos de memória transacional [HMPJH05]. A utilização destes mecanismos permite evitar, mas não garante, a inexistência de interferências, excepto nos sistemas de memória transacional, que garantem esse tipo de propriedades.

Uma *race condition* acontece quando a noção de correcção de um programa, mais ou menos precisa, está dependente da maneira como as *threads* intercalam a execução umas com as outras e do momento em que o fazem. Uma má interacção entre *threads* pode produzir resultados errados e/ou inesperados. Considere-se o programa concorrente em que duas *threads* executam o código `i++` para incrementar uma variável partilhada (`i`). Dado que as operações de leitura do valor da variável podem não ser executadas sem que haja uma mudança de contexto, é possível encontrar um entrelaçamento de execuções de programas que produza um resultado inesperado (ver figura 1.1). Este tipo de interferência é conhecida como uma *race condition* e acontece quando dois *threads* acedem em simultâneo à mesma zona de memória. No exemplo da figura, o valor final esperado na variável `i` é `2`, no entanto, devido à interferência nesta sequência particular de operações elementares, o valor final é `1`.

Em programas com memória partilhada a solução para evitar este tipo de problemas é utilizar mecanismos dinâmicos de controlo de concorrência, o que exige uma maior disciplina de programação, por forma a permitir que as várias *threads* concorrentes possam partilhar as mesmas regiões de memória. Este tipo de abordagens é muito sensível aos erros de programação e dependem bastante do cuidado de quem cria os programas.

Uma abordagem possível, mas pouco utilizada, para garantir a correcta utilização dos mecanismos de controlo de concorrência é a utilização de verificação estática de código. A verificação estática não faz controlo directo de concorrência, apenas se destina a verificar a correcta

utilização dos mecanismos de controlo de concorrência. As abordagens gerais que usam verificação estática para controlo de concorrência vão desde os trabalhos iniciais baseados em lógica de Hoare [Hoa69] aplicando as operações de monitores [Hoa74], até aos mais recentes sistemas de verificação de *ownership* e *aliasing* [CD02, SD03, CDNS07] e ainda às linguagens orientadas por objectos com suporte para *typestates* [ASSS09], baseadas em tipos espaciais [Cai08] e lógica de separação [Bor10].

Objectivos Neste trabalho implementa-se uma abordagem de verificação estática do controlo de concorrência baseada em tipos, inspirada nos trabalhos de Luís Caires et al. [Cai08, Mil08, CS10]. Utilizam-se aqui os tipos espaciais comportamentais [Cai08], para definir protocolos de utilização tal como em [Mil08], que são utilizados para detectar interferências entre *threads* e disciplinar o acesso a zonas de memória partilhada. Pretende-se através desta técnica implementar uma linguagem de programação com mecanismos para lidar com concorrência, onde é verificada estaticamente, através do seu sistema de tipos, a ausência de *race conditions*. Alguns aspectos práticos da linguagem implementada são inspirados na linguagem *Featherweight Java* (FWJ) [IPW99], nomeadamente a sintaxe das classes, a declaração das variáveis de instância e a estrutura dos construtores. Acrescentam-se as construções correspondentes a linguagens imperativas, tais como a declaração das variáveis de instância, a expressão de afectação, ciclos *while*, etc. Para além disso a linguagem não tem herança e a comparação de tipos é estrutural.

A linguagem implementada permite que os objectos possam ser compostos a partir de outros objectos existentes, desde que fornecidos na altura da instanciação. As variáveis do tipo "objecto" estão associadas a interfaces que são definidos como tipos comportamentais, com protocolos de execução próprios. De maneira a controlar o grau de dificuldade do trabalho, limita-se a operação de afectação às variáveis de instância de tipos básicos. O tratamento de afectações a objectos (com tipos comportamentais) requer uma compreensão bastante mais profunda de noções de *ownership* dos objectos, o que cai necessariamente fora do âmbito deste trabalho. Contudo, apesar desta restrição, o estado dos objectos efectivamente muda com a afectação das variáveis de instância, sendo ainda assim possível construir exemplos interessantes.

Exemplo Ilustra-se agora a linguagem proposta, através de pequenos exemplos incrementais. Descrevem-se as principais expressões da linguagem, bem como a utilização do sistema de tipos para disciplinar a utilização de objectos e evitar *race conditions*. Considere-se a seguinte definição para a classe `CellImpl`

```
class CellImpl(elem:int) {
  void set(n:int) { elem = n }
  int get() { elem }
}
```

Note-se que o cabeçalho da declaração da classe `class CellImpl(elem:int)` agrega a declaração das variáveis de instância e a declaração do único constructor. Assim, os objectos da classe `CellImpl` têm uma variável de instância de tipo inteiro chamada `elem` que é inicializada com o argumento do construtor. Da definição constam ainda as declarações de métodos, onde se definem os parâmetros, o tipo de retorno e a expressão que implementa cada método. A linguagem possuiu expressões de declaração de identificadores (**`let`**), expressões condicionais (**`if`**), ciclos (**`while`**), bem como as operações habituais sobre valores inteiros e boleanos (`+`,`*`,**`and`**, **`not`**, ...).

Introduziu-se na linguagem a expressão que denota a avaliação sequencial de duas expressões (`;`) e a expressão que denota a execução paralela de duas expressões (`|`). Desta maneira é possível a construção de programas concorrentes como o exemplo seguinte, onde o método `get` é chamado duas vezes em paralelo.

```
let x = new CellImpl(1) in { x.get() | x.get() }
```

As expressões da forma e_1 `|` e_2 permitem definir programas com vários fluxos de execução concorrentes de uma maneira estruturada através de mecanismos de *fork* e *join* (*wait*), em oposição à criação livre de *threads*. A expressão que primeiro terminar a sua execução fica em espera até que a outra expressão também termine para o programa poder continuar a sua execução. O resultado de uma expressão e_1 `|` e_2 é sempre o valor **null** e o tipo correspondente é o tipo **void**.

Existem algumas linguagens de programação que incluem abstracções de programação para concorrência, tais como o Concurrent Pascal [Han76] ou o Concurrent ML [Rep93] (ver capítulo 2). Neste trabalho é apresentada uma linguagem com abstracções para concorrência, em que é possível a verificação de um programa detectar *race conditions* a partir de tipos comportamentais, definidos em protocolos de utilização, tratados por um algoritmo de tipificação próprio.

Nem sempre a definição de vários traços de execução simultâneos resulta num programa correcto. Por exemplo, uma expressão como `x.get()|x.set(2)` pode causar uma interferência entre *threads*, entre uma escrita e uma leitura da mesma variável. É portanto necessário disciplinar a utilização de métodos e variáveis entre os diferentes fluxos de execução.

Interfaces dos Objectos Para disciplinar a chamada de métodos utilizam-se tipos comportamentais na descrição dos objectos [Cai08]. Os tipos dos objectos (interfaces) são declarados com informação adicional acerca do modo como os seus métodos podem ser chamados. Este tipo de anotações aparece em vários outros trabalhos, como por exemplo [Mil08]. Estende-se aqui essa abordagem com a introdução de abstracções para concorrência, inspirada em [Cai08]. Considere-se a interface `Cell` definida da seguinte maneira:
A interface `Cell`, para além de descrever o tipo dos seus métodos, descreve também a forma como estes podem ser chamados. Neste caso, "**`usage`** `set;get`" significa que a execução dos métodos `set` e `get`, tem que ser feita por essa mesma ordem. Após a execução destes dois

```
interface Cell {
  void set(n:int);
  int get();

  usage{ set;get }
}
```

métodos, não é possível executar mais nenhuma operação com o objecto. Declara-se que uma classe implementa uma interface da seguinte maneira:

```
class CellImpl(elem:int) implements Cell { ... } (..)
```

No caso da expressão

```
let x = new CellImpl(1) in { x.set(2); x.get() };
```

o protocolo do objecto associado ao identificador `x` é respeitado. Já na expressão

```
let y = new CellImpl(1) in { y.get(); y.set(2) }
```

a chamada de métodos de `y` viola o protocolo estabelecido. Aos objectos que implementam a interface `Cell` não é permitido executar o método `get` antes do método `set`.

A definição dos protocolos de utilização é um mecanismo que permite ao programador definir a ordem de chamadas a métodos em cada momento de execução. A verificação das classes apoia-se no protocolo de utilização. A utilização dos vários campos de um objecto é combinada de acordo com o protocolo definido na interface, permitindo reconhecer interferências indesejadas na definição dos métodos e/ou do protocolo de utilização.

Recorrendo mais uma vez ao exemplo da interface `Cell`, se o protocolo definido referisse uma possibilidade de utilização paralela do método `set`, a classe estaria mal tipificada em relação ao protocolo definido, `(set;get)`. Poder-se-ia incorrer numa situação de *race condition*, provocada por duas "escritas" em paralelo sobre a mesma zona de memória partilhada (a variável `elem`).

Tipos para Variáveis Actualmente, o sistema de tipos aqui apresentado apenas permite afectações para variáveis de instância de tipo básico (inteiros, boleanos e *strings*). Utilizam-se diferentes notações (e tipos) para constantes, variáveis e tipos encontrados em métodos sincronizados. Em particular, uma expressão do tipo inteiro (**int**) pode ser tipificada com as seguintes variantes:

- **int** – Constante do tipo inteiro ou utilização de uma variável como constante,
- **ref**(**int**) – Variável do tipo inteiro. Tem-se a certeza que um identificador é do tipo **ref** quando esse identificador é objecto de uma afectação, ou é uma variável de instância,
- **sync**(**int**) ou **sync**(**ref**(**int**)) – Constante ou variável do tipo inteiro utilizada dentro de um método sincronizado.

Tome-se o exemplo de um mesmo identificador `x` em duas afectações em paralelo:

```
x := 1 | x := 2
```

Pode facilmente verificar-se que, considerando por hipótese a afectação uma operação não atómica, a execução paralela de duas afectações para o mesmo identificador incorre numa interferência. Para verificar a inexistência de interferências a linguagem recorre aos tipos das variáveis e à utilização que lhes foi dada em cada expressão. No caso da expressão `x := 1 | x := 2`, tem-se:

$$x : \mathbf{ref}(\mathbf{int}) \mid \mathbf{ref}(\mathbf{int})$$

As duas afectações em paralelo são identificadas pelos dois tipos "referência" (**ref**), denotando uma utilização incompatível do identificador `x`. Considerem-se agora outros dois casos:

```
x := 1 ; x := 2
x := 1 | y := 2
```

Tanto numa situação (`x := 1 ; x := 2`) como noutra (`x := 1 | y := 2`) não existe qualquer possibilidade de interferência. Os tipos dos identificadores são:

$$x : \mathbf{ref}(\mathbf{int}) \; ; \; \mathbf{ref}(\mathbf{int})$$

$$x : \mathbf{ref}(\mathbf{int}) \; , y : \mathbf{ref}(\mathbf{int})$$

Se forem feitas duas afectações para o mesmo identificador `x` (x : **ref**(**int**) ; **ref**(**int**)) numa sequência, não há possibilidade de haver interferências. As afectações ocorrem necessariamente em momentos distintos. Duas afectações em paralelo, para dois identificadores diferentes (`x := 1 | y := 2`) também não levantam qualquer problema. A utilização dos dois identificadores (x : **ref**(**int**) , y : **ref**(**int**)) também é considerada válida. O tipo **sync** é introduzido na tipificação dos métodos sincronizados. Os métodos sincronizados restringem o acesso às variáveis de instância a uma *thread* de cada vez. Sempre que há mais do que uma *thread* a tentar aceder às variáveis de instância, se todas as *threads* correspondem à chamada de métodos sincronizados, apenas uma delas tem acesso às variáveis de instância. Todas as outras ficam bloqueadas até que chegue a sua vez. Assim, se todos os métodos que acedem a uma variável de instância forem sincronizados, essa variável de instância estará bem tipificada.

$$\mathbf{sync}\ m_1\ \{\ \ldots\ \texttt{x := 1}\ \ldots\ \} \mid \mathbf{sync}\ m_2\ \{\ \ldots\ \texttt{x + 1}\ \ldots\ \}$$

No exemplo acima apresentam-se dois métodos sincronizados, m_1 e m_2 que utilizam o mesmo identificador `x` de uma classe que apenas tem esses dois métodos. Considerando `x` uma variável de instância, não haveria qualquer problema em utilizar os dois métodos em paralelo. São identificadas uma escrita em m_1 (x : **ref**(**int**)) e uma leitura em m_2 (x : **int**).

Como os métodos são sincronizados, para uma utilização paralela dos métodos ($m_1 \mid m_2$), o identificador `x` tem o seguinte tipo:

$$x : \mathbf{sync}(\mathbf{ref}(\mathbf{int})) \mid \mathbf{sync}(\mathbf{int})$$

Como os métodos m_1 e m_2 são sincronizados, o sistema irá gerir o acesso a memória de maneira a que o identificador `x` não seja utilizado pelo método m_1 durante a execução do método m_2, ou por m_2 durante a execução do método m_1.

Operadores de Escolha, Replicação e Repetição Para enriquecer as possibilidades de exprimir protocolos de utilização de objectos, a linguagem implementa ainda operadores de escolha (`&`), repetição (`*`) e replicação (`!`) presentes em [Cai08].

Uma expressão da forma e_1 & e_2 não restringe a ordem de execução, mas assim que um dos ramos é escolhido, o outro deixa de poder ser executado. O operador "`!`" permite que o mesmo método seja executado em paralelo um número indeterminado de vezes e o operador de repetição "`*`" permite que o mesmo método seja executado em sequência um número indeterminado de vezes. Considere-se a interface

```
interface CellChoice {
  void set(n:int);
  int get();

  usage{ set & get }
}
```

onde `usage set & get` indica que se pode chamar uma vez o método `set` ou chamar uma vez o método `get` mas não se podem chamar os dois métodos, seja em sequência ou paralelo. Apesar de introduzir mais possibilidades na ordem de execução, o protocolo definido em `CellChoice` restringe o número de métodos que é possível chamar. O protocolo de escolha apenas permite que se possa executar um dos lados (neste caso métodos) do operador, mas sem qualquer ordem específica. No programa seguinte:

```
interface CellGet {
  int get();
  usage{ get! }
}
class CellImpl(elem:int) implements CellGet {
  int get() { elem }
}
main {
  let x = new CellImpl(1) in {
    (x.get()|x.get()) | (x.get();x.get())
  }
}
```

o método `get` de um objecto que implemente `CellGet` pode ser chamado várias vezes em paralelo. A expressão "`(x.get()|x.get()) | (x.get();x.get())`" é uma utilização válida de um objecto do tipo `CellGet`.

Combinando os operadores de escolha e replicação (`&` e `!`), é possível especificar uma utilização mais rica para uma célula de memória: **`usage`**`{ set&get! }`. Partindo deste protocolo, já será possível "executar vários `get` em simultâneo ou um único `set`".

Este protocolo é ainda algo limitativo, mas com a introdução do operador de repetição (`*`) pode obter-se um novo protocolo, próximo do desejado: **`usage`**`{ (set&get!)* }` Com este novo protocolo é possível "executar vários `get` em simultâneo ou um único `set`, seguidos (ou não) de sequências desse mesmo tipo de execuções", que permite que uma célula seja escrita e lida várias vezes. Será assim possível executar uma expressão como `(get|get);set;set;get`.

Com os vários operadores apresentados ao longo deste capítulo conseguem exprimir-se alguns protocolos de utilização interessantes, que dotam a linguagem de uma boa capacidade de adaptação aos mais variados casos.

Acesso a Memória Partilhada A chamada de operações não atómicas para a mesma zona de memória partilhada pode levar a valores não esperados e que afectam a correcção de um programa [Rey78, GBB+06]. Apesar de um programa não incorrer num erro de execução, o resultado final pode ser imprevisto.

O modelo de memória partilhada adoptado neste trabalho é limitado aos tipos básicos, definidos nas variáveis de instância. Segue-se uma disciplina de *one writer/many readers* e um programa só se considera bem tipificado se respeitar este modelo para todas as variáveis. A garantia de acessos (bem) disciplinados às variáveis é conseguido tipificando o código das classes em relação ao tipo das suas variáveis de instância, protocolos ou variáveis básicas. No caso do exemplo apresentado ao longo deste capítulo, o código da classe é tipificado em relação à variável `elem`, do tipo **int**.

Um identificador pode ser utilizado tantas vezes quantas necessárias ao longo de um programa, ou de uma classe, desde que não haja afectações em expressões paralelas para o mesmo identificador. Por exemplo, para um identificador que sofra uma afectação e de seguida, em sequência, seja utilizado como constante, denota uma utilização **ref**(**int**); **int**. Esta utilização é considerada válida desde que o identificador seja uma variável de instância, podendo por isso ser afectado.

1.1 Algoritmo de Tipificação

Nesta dissertação apresenta-se um algoritmo de tipificação com base em inferência de tipos, que analisa as expressões de um programa para reconhecer qual a utilização feita dos vários

identificadores. A inferência de tipos constrói um conjunto de restrições que são depois comparadas por *subtyping*, com o verdadeiro tipo dos identificadores.

Subtyping A relação de *subtyping* baseia-se no princípio da substitutividade e, no caso concreto dos objectos, esse princípio é aplicado no protocolo de utilização dos objectos, ou seja, nos métodos que cada objecto pode executar em cada momento. A comparação não se faz por qualquer nome de classe ou hierarquia de classes explicitamente definida, mas sim através da estrutura dos tipos (protocolos). Por exemplo, considerando os tipos $\tau = get \mid get$ e $\sigma = get \ ; \ get$. Pode verificar-se que $\tau <: \sigma$ (τ é subtipo de σ). O inverso não se verifica pois o tipo σ está limitado a uma utilização sequencial dos métodos. Esta verificação é feita de maneira recursiva no protocolo do tipo, comparando o conjunto de métodos que podem ser chamados a cada momento. Chama-se a esse multiconjunto de métodos a "frente" do tipo do objecto. Por exemplo, a frente de τ é $\{get, get\}$. Já a frente de σ é $\{get\}$ e apenas tem uma ocorrência do método get. A partir da análise das frente dos tipos podemos concluir que, no mesmo momento de execução (neste caso o inicial), τ pode executar mais operações que σ, ou seja, a substituição de τ por σ não seria compatível. Essa propriedade deve ser mantida ao longo do comportamento definido pelos tipos.

A relação de *subtyping* entre tipos de variáveis básicas é usada de forma especial, pois são as únicas que podem ser afectadas com novos valores. Um variável do tipo **int** pode ser usada e afectada em sequências de expressões. Por exemplo, **ref**(**int**); **int** é uma utilização correcta. Já a utilização em expressões em paralelo não deve ser permitida e uma expressão com o tipo **ref**(**int**) | **int** para o mesmo identificador indica a presença de um erro, devido a uma escrita e uma leitura em simultâneo.

Inferência de Tipos (Utilização de Objectos) Através de um algoritmo de inferência é possível extrair a utilização feita dos identificadores livres em cada expressão e compará-la com os tipos (protocolos de utilização) definidos, para garantir que foram usados de acordo com a sua especificação. Implementa-se uma técnica clássica de Damas Milner [DM82], combinada com um algoritmo de *subtyping* [Sta88, Mit91].

Na declaração de identificadores filtram-se os tipos inferidos para serem comparados com os tipos definidos pelo algoritmo de tipificação. Inicialmente são filtradas todas as declarações locais, depois os parâmetros dos métodos e por fim as variáveis de instância das classes. O algoritmo termina quando já não existirem identificadores livres e se todos os identificadores foram usados tal como definidos, o programa considera-se correctamente tipificado.

Tome-se como exemplo o método `void set(n:int) { elem := n }`. Na expressão `elem := n` ocorrem dois identificadores livres: `elem` e `n`. Como ambos são identificadores livres, atribui-se uma variável de tipo (X) ao identificador `elem`. No corpo do método, `elem` é objecto de uma afectação, logo o tipo desse identificador será uma referência para o tipo X, ou seja **ref**(X). Ao analisar o lado direito da expressão da afectação encontra-se outro identificador livre (`n`), ao

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n }
  int get() { elem }
}
```

Listagem 1.1: Célula de memória

qual se atribui outra variável de tipo, Y. Da expressão da afectação resulta ainda a restrição $Y <: X$ correspondente à relação entre os identificadores `elem` e `n`:

$$\texttt{elem}\text{: } \mathbf{ref}(X)$$
$$\texttt{n}\text{: } Y$$
$$Y <: X$$

Depois de tipificar o corpo do método deve garantir-se que os identificadores livres foram usados correctamente. Neste caso, o método tem o parâmetro `n` do tipo **int**. Logo, adiciona-se a restrição:

$$\mathbf{int} <: Y$$

Como o conjunto de restrições $\mathbf{int} <: X$ e $Y <: X$ é consistente, encerra-se a tipificação do método com a informação de que a variável `elem` foi usada com o tipo $\mathbf{ref}(X)$ e a verificação do método adicionou as restrições $\mathbf{int} <: Y$ e $Y <: X$.

No caso da classe `CellImpl` (listagem 1.1), tendo em conta o protocolo de utilização definido na interface `Cell`, resulta o seguinte tipo para a variável `elem`:

$$\texttt{elem}\text{: } (\mathbf{ref}(\mathbf{int}) \ \& \ \mathbf{int}!)*$$

que corresponde à junção dos tipos das variáveis tal como utilizadas nos métodos, substituindo esses tipos tal como organizados no protocolo da interface. Como não existem possíveis interferências entre escritas e leituras para a variável `elem` (não há tipos $\mathbf{ref}(\mathbf{int})$ em paralelo), a utilização dessa variável é considerada válida e a definição da classe considerada correcta, compatível com o tipo $\mathbf{ref}(\mathbf{int})$ do campo `elem`.

Algoritmo de Simulação O algoritmo de tipificação depende, em alguns pontos, da verificação da compatibilidade entre a utilização de um objecto e o protocolo com o qual foi definido. Para verificar essa compatibilidade é utilizado um algoritmo de simulação entre tipos.

```
let x = new CellImpl(1) in {
  out(x.get());
  x.set(5);
  out(x.get())
}
```

No exemplo acima, é definido um identificador `x` associado a uma instância da classe `CellImpl`. Ao ser associado à classe `CellImpl`, o identificador `x` está obrigado a respeitar o protocolo de utilização definido na interface `Cell`. A partir da expressão `out(x.get());x.set(5);out(x.get` conclui-se que a utilização associada ao valor `x` corresponde ao seguinte tipo[1]:

$$get; set; get$$

Sendo que o protocolo definido na interface `Cell` é `usage{(set&get!)*}`, é possível estabelecer uma relação de *subtyping* entre os dois tipos:

$$(set\&get!)* <: get; set; get$$

através da simulação de um tipo no outro. A ideia geral do algoritmo é que um tipo τ é subtipo de um tipo σ se τ conseguir efectuar todas as transições de σ. Nesse sentido, os vários métodos do supertipo σ vão sendo "consumidos" (chamadas de métodos) e a simulação prossegue:

$$\begin{aligned}
&(set\&get!)* <: get; set; get\\
&(set\&get!)* <: set; get\\
&(set\&get!)* <: get\\
&(set\&get!)* <: \mathsf{stop}
\end{aligned}$$

até se chegar a um caso terminal: $(set\&get!)* <: \mathsf{stop}$. O operador de repetição ($*$) permite a execução zero ou mais vezes para um tipo, logo o tipo $(set\&get!)*$ pode não ser executado e, como tal, substituir o tipo stop. Neste ponto, o algoritmo de simulação termina e como conseguiu fazer todas as transições, logo o programa é considerado como bem tipificado.

1.2 Principais Contribuições

A finalidade desta dissertação é disponibilizar um protótipo de uma linguagem de programação com suporte para verificação estática de controlo de concorrência e respectivo algoritmo de tipificação. É esperado que o sistema implementado possa contribuir de forma satisfatória, como uma forma de verificar estaticamente o controlo de concorrência, permitindo precaver a existência de erros de execução em programas algo complexos e em que, muitas vezes, é difícil para o programador efectuar essa mesma verificação.

[1] Simplifica-se o tipo ignorando os tipos de retorno e parâmetros dos métodos.

Neste documento são definidas e apresentados os algoritmos de tipificação para a linguagem (FWCJ – Featherweight Concurrent Java). O interpretador da linguagem tem por base um sistema de tipos espaciais-comportamentais verificado através de um algoritmo de tipificação. O algoritmo de tipificação usa um misto de verificação e inferência de tipos, de modo a que os programas não tenham que ser excessivamente anotados e, ainda assim, permitir a verificação do mesmo. Para além disso, o algoritmo de tipificação faz uso de um algoritmo de *subtyping* que relaciona, de uma forma bastante rica, diferentes protocolos de utilização de objectos. O algoritmo de *subtyping* implementa uma relação de simulação entre tipos segundo o princípio geral do *subtyping* comportamental (ver [LW94]).

Não é feita qualquer prova de correcção para o algoritmo implementado, no entanto, o mesmo é validado por uma implementação concreta da linguagem. A implementação (um protótipo) permite verificar uma considerável bateria de testes, para os mais variados casos e, espera-se, dar uma boa indicação da correcção do algoritmo.

1.3 Estrutura do Documento

Depois desta primeira introdução segue-se para o capítulo 2, onde é feita uma descrição do "estado da arte" relacionado com o tema desta dissertação. São abordadas várias técnicas para lidar com concorrência, algumas das quais servem de base para o trabalho desenvolvido. No capítulo 3 é descrita a linguagem implementada e o respectivo algoritmo de tipificação. Por fim, no capítulo 4 discutem-se os resultados e dificuldades encontradas ao longo da realização da dissertação e são abordados alguns pontos que poderiam ter sido mais desenvolvidos ou outros que possam ser objecto de estudo no futuro.

2

Trabalho Relacionado

O trabalho descrito nesta dissertação está relacionado com a problemática de evitar *race conditions* em programas concorrentes. Para enquadrar o trabalho, apresenta-se uma breve descrição dos principais mecanismos de controlo de concorrência e as linguagens onde esses mecanismos são mais relevantes. São também abordados trabalhos que introduzem mecanismos diversos para verificar estaticamente anomalias relacionadas com concorrência, nomeadamente através do *aliasing* de referências. Por fim, são estudadas algumas técnicas que utilizam tipos comportamentais para lidar com concorrência, abordagem que servirá de base para o trabalho desta dissertação.

2.1 Linguagens com Suporte para Concorrência

Existem inúmeras linguagens com abstracções próprias para lidar com concorrência, que se podem dividir em duas categorias diferentes: as baseadas em memória partilhada e as baseadas em troca de mensagens.

Troca de Mensagens Uma das linguagens que suporta concorrência através de troca de mensagens é o ConcurrentML [Rep93]. O ConcurrentML é uma linguagem multi-paradigma (funcional e imperativa) que funciona à base de eventos, em que estes são tratados concorrentemente através de troca de mensagens. Outro exemplo é a linguagem Erlang [AVWW93], que segue um modelo de programação através de "actores" [Agh85] para gerir processos concorrentes. Os actores funcionam através de instruções passadas por mensagens específicas a que definem o comportamento que o actor deve tomar após a sua recepção. A linguagem

Scala [OSV08] também segue este tipo de abordagem, baseada em *pattern matching* e actores.

Podem ser seguidas outras abordagens como a seguida pela linguagem Go! [CM04], em que a concorrência é tratada por "agentes". A ideia da utilização de agentes é que estes sejam comandados pelos vários processos em execução e, a partir de mensagens, sejam enviados para outros processos onde possam executar outras tarefas.

Concorrência com Memória Partilhada A linguagem ADA [Pyl85] foi uma das primeiras linguagens a suportar concorrência através de memória partilhada. A linguagem deriva da linguagem Pascal e os programas consistem em pacotes, funções e procedimentos. Para tratar a questão da memória partilhada é usada uma construção própria da linguagem (`task`) onde é definida a forma como as várias *threads* de um programa podem interagir. O Concurrent Pascal [Han76], antecessor do ADA, é também outra linguagem que se baseia no Pascal e utiliza monitores para controlar a concorrência. Existem ainda mais algumas linguagens que seguem este tipo de abordagem, como o Modula-3 [Nel91] e fazem a gestão de memória partilhada através de monitores, com *wait* e *signal*. Mais recentemente a linguagem Java e a linguagem C# seguem este modelo.

Outras linguagens como o Cω [BMS05], desenvolvida pela Microsoft Research e que estende a linguagem C#, ou o Join Java [IK02] que se baseia na linguagem Java, apresentam mecanismos que suportam os dois tipos de abordagens, tanto para memória partilhada como para troca de mensagens.

2.2 Controlo de Concorrência

O controlo de concorrência ao nível das linguagens de programação pode ser feito utilizando directamente mecanismos de baixo nível (semáforos, *locks*, etc.), ou recorrendo a abstracções que utilizem esse tipo de mecanismos para gerir e sincronizar o acesso a zonas críticas de um programa.

2.2.1 Semáforos e *Locks*

Os semáforos e os *locks* são dois mecanismos utilizados para controlar o acesso a regiões críticas de um sistema.

Os *locks* para exclusão mútua funcionam através de primitivas de bloqueio (*lock*) e desbloqueio (*unlock*) para uma dada região crítica. Sempre que uma *thread* quer actuar sobre uma região crítica, executa a operação *lock* para poder operar sobre essa região e bloquear o acesso a todas as outras *threads*. Quando terminar as operações sobre a região crítica, a *thread* deve libertar o acesso a essa região, através da operação *unlock*, para que outras *threads* lhe possam aceder através do mesmo processo. A figura 2.1 apresenta um exemplo de duas *threads* a tentar aceder à mesma região crítica. Podem ainda ser utilizados `locks` de exclusão mútua (*mutexs*)

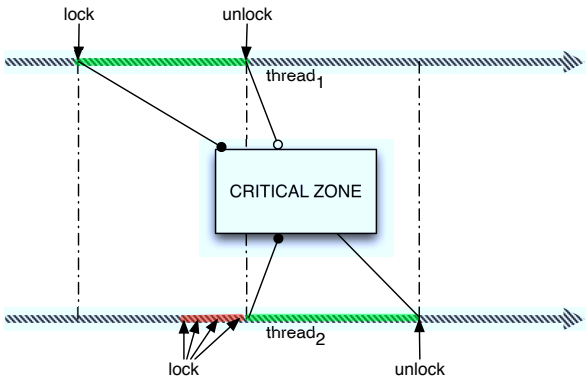

Figura 2.1: Duas *threads* a utilizar *locks* para aceder a uma região crítica

que bloqueiam o acesso a uma região crítica ao nível do sistema operativo, ao contrário dos *locks* que apenas bloqueiam ao nível da aplicação.

Os semáforos introduzidos por Dijkstra [Dij65, Dij75] consistem numa variável inteira que, através das operações atómicas P e V controla os acessos a regiões críticas. Essas operações são definidas da seguinte forma:

```
V (int sem) { sem := sem +1 }
P (int sem) { repeat: sem > 0 ? sem := sem - 1 ; break }
```

Tome-se o exemplo de múltiplos leitores e escritores para a mesma zona memória em que se segue uma disciplina de *many readers/one writer*:

```
int sem := 1;
int semReaders := 1;
int readers := 0;

//Readers
P(semReaders);
readers++;
if(readers == 1) P(sem);
V(semReaders);
//read data...
P(semReaders);
readers--;
if(readers == 0) V(sem);
V(semReaders);
```

Para os leitores, e em específico para o primeiro este deve bloquear o acesso para leitura (`P(sem)`). Para ser possível saber se há leitores em execução utiliza-se um segundo semáforo (`semReaders`) para controlar o número de leitores activos. Depois de ler os dados, o leitor decrementa o número de leitores por exclusão mútua através do semáforo `semReaders` e, se for o último, liberta o acesso para o semáforo da região crítica (`V(sem)`) possibilitando que escritores ou novos leitores possam aceder ao recurso partilhado.

```
boolean CAS(ref item, expectedValue, newValue){
  if (item != expectedValue)
    return false;
  item = newValue;
  return true;
}
```

Listagem 2.1: Algoritmo de *compare and swap*.

```
do {  a = i; } while( !CAS(&i, a, a+1) )
```

Listagem 2.2: Algoritmo não bloqueante para incremento de uma variável partilhada

```
//Writers:
P(sem)
//write data...
V(sem)
```

Como seria de esperar, os escritores fazem o seu acesso por exclusão mútua a partir do semáforo `sem`, partilhado com os leitores.

2.2.2 *Compare and Swap*

A utilização da operação atómica *compare and swap* (CAS) [MS96], um mecanismo primitivo dos sistemas de operação, é outra das técnicas utilizadas para resolver problemas de concorrência. A técnica consiste na verificação e troca do valor de uma variável através de uma operação atómica CAS, cuja semântica é dada pelo código da listagem 2.1.

A função CAS tem três argumentos: uma referência para a posição em memória da variável a ser alterada, o valor que se espera que essa variável apresente imediatamente antes de trocar para o novo valor e o valor depois de ser efectuada a alteração. A troca só é efectuada se o valor da variável se tiver mantido.

A verificação da interferência com outra *thread* antes da entrada na operação atómica de CAS, é feita através do valor esperado e da referência para o valor actual em memória. Voltando ao exemplo apresentado no capítulo 1 (figura 1.1), seria possível ao *thread* t_2 verificar que já tinha sido feita uma alteração ao valor de `i` através da junção da técnica de CAS (listagem 2.1), usada na listagem 2.2. Através desta operação, seria possível ao *thread* t_2 verificar a *race condition*, pois estaria à espera de um valor anterior de `i` (`i = 0`), quando esse valor já tinha sido alterado para `i = 1`. Como tal, seria novamente executada a operação, sem prejuízo no resultado final. A operação CAS é usada nos chamados algoritmos não bloqueantes e não existem ainda técnicas para verificar estes algoritmos.

2.2.3 Monitores

Uma das técnicas empregues no controlo de concorrência é a utilização de monitores [Hoa74, Hoa78, Han93]. Os monitores introduzem novas possibilidades de controlo ao representarem zonas de código com acesso exclusivo no programa. A exclusividade no acesso a estas zonas é enriquecido pela utilização de condições (`wait` e `signal`), que permitem que uma *thread* passe a vez a outra voluntariamente. Se uma *thread* não conseguir adquirir o acesso à região crítica de memória protegida, entra em espera até que outra *thread* lhe dê permissão para operar nessa região.

Para cada bloco de código que se queira proteger, é criada uma variável de condição na qual se pode esperar por um sinal – `wait` – ou enviar um sinal – `signal` – para passar o controlo a outra *thread*.

2.3 Verificação Estática

2.3.1 Lógica de Hoare

A lógica de Hoare é um sistema dedutivo para raciocinar sobre determinadas propriedades dos programas e baseiam-se nos chamados triplos de Hoare [Hoa69]. Um triplo de Hoare é definido por:

$$\{A\}P\{B\}$$

onde A e B são duas fórmulas que correspondem, respectivamente, às pré-condições e pré-condições de um programa P. P é considerado um programa válido se, quando começa a sua execução num estado que satisfaz A e termina, então termina num estado que satisfaz B. Siga-se a regra para a afectação definida num triplo de Hoare:

$$\{A(x/E)\}\ x := E\ \{A\}$$

onde $A(x/E)$ significa que todas as ocorrências livres de x em A são substituídas por E. A partir desta regra conclui-se que o triplo:

$$\{x > -1\}x := x + 1\{x > 0\}$$

define que num programa em que, previamente, x é maior que -1 e lhe é adicionado uma unidade, então x passa a ser maior que 0.

Lógica de Hoare aplicada a Monitores As operações sobre variáveis de condição podem ser associados a regras de lógica de Hoare [Hoa69], o que permite raciocinar sobre programas concorrentes:

$$\{true\}wait(C)\ \{cond(C)\}$$

```
single resource:monitor
begin busy:Boolean;
    nonbusy:condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
          busy := true;
    end;
  procedure release;
    begin busy := false;
        nonbusy.signal
    end;
  busy := false; comment initial value;
end single resource
```

Listagem 2.3: Exemplo do funcionamento de monitores (Pascal)

```
procedure acquire;
  begin while busy then nonbusy.wait;
            busy := true;
  end;
```

Listagem 2.4: Processo de espera com comando while

$$\{cond(C)\}\ signal(C)\{true\}$$

em que $cond(C)$ significa que é satisfeita a propriedade associada à variável de condição C. A listagem 2.3 (retirada de [Hoa74]) ilustra a aquisição e libertação do acesso a um recurso para um determinado bloco de instruções (neste caso apenas é alterado o estado para "ocupado").

Sempre que um procedimento entra no monitor, coloca o estado como ocupado (`busy := true`). Sempre que um novo *thread* tenta adquirir o *lock*, depara-se com o estado ocupado e entra em espera (`nonbusy.wait`) até que o *thread* anterior liberte (`nonbusy.signal`) e passe a assinalar o estado livre.

Existem modelos diferentes para reagir a uma instrução `nonbusy.signal`. Em algumas linguagens apenas um *thread* em espera é "acordado" e, nesse caso o exemplo anterior mantém-se válido. No entanto, existem linguagens em que a instrução para libertar o *lock* provoca o "acordar" de todos os *threads*. Nestes casos tem que ser feita uma adaptação do processo para adquirir o *lock* (*notifyAll*).

Usando esta abordagem (tal como na listagem 2.4), não se corre o risco de que vários *threads* possam alterar a variável `busy` em simultâneo, continuando a garantir o bom funcionamento de todos os *threads*.

2.3.2 *Extended Static Checker for Java*

O ESCJava [FLL+02, RLNS00] é uma das ferramentas através da qual se pode estender a verificação estática do Java (verificação de tipos) para uma verificação que cubra mais propriedades. Existem outras da mesma categoria, como o Spec# [BLS05] ou o SpecJava [San10].

```
public class Account {

  //@ invariant balance >= 0;
  private int balance;

  //@ requires v >= 0;
  //@ ensures balance == v;
  //@ modifies balance;
  public Account(int v){
    balance = v;
  }

  //@ requires x >= 0;
  //@ ensures balance == \old(balance) + x;
  //@ modifies balance;
  public void credit(int x){
    balance += x;
  }

  //@ requires balance >= x;
  //@ requires x >= 0;
  //@ ensures balance == \old(balance) - x;
  //@ modifies balance;
  public void debit(int x){
    balance -= x;
  }

  //@ ensures \result >= 0;
  //@ ensures \result == balance;
  public /*@ pure @*/ int balance(){
    return balance;
  }
}
```

Listagem 2.5: Especificação em ESCJava para uma conta bancária.

Através do seu sistema de anotações feitas na linguagem JML [LBR98] (*Java Modeling Language*), o ESCJava permite verificar estaticamente erros que poderiam acontecer em tempo de execução, tais como referências nulas, acesso a posições de *arrays* inexistentes, ou até erros de *type cast* em variáveis. Este sistema permite também inferir propriedades de sincronização, tais como a ausência de *deadlocks* e *race conditions*. Através de anotações no código Java, o ESCJava pode também ser utilizado para especificar os programas a desenvolver, mesmo antes dos seus métodos serem definidos, para garantir atempadamente que a sua definição irá atingir os objectivos pretendidos.

Ao contrário de outros sistemas de verificação estática o ESCJava efectua verificação de forma *modular*, ou seja, não precisa de conhecer todo o sistema para verificar apenas uma parte do mesmo (método ou classe). Por exemplo, para a classe "conta" (`Account`) definida na listagem 2.5, poderia apenas ser verificado o método `credit`, para o qual, além das três anotações específicas desse método, apenas seria verificada a invariante da classe.

A ferramenta ESCJava traduz o corpo de cada método para ser verificado com base em *guarded commands* (GCs), definidos por Dijkstra [Dij75], que verifica a correcção de programas através do uso de lógica de Hoare. O ESCJava gera as pré-condições mais fracas (*verification conditions* – VCs) para cada uma das traduções anteriores, garantindo que as proposições geradas pelo *translator* possam ser verificadas. O passo seguinte é efectuar a verificação com base nas VCs geradas nos passos anteriores. O resultado são informações passadas ao utilizador sobre eventuais erros e/ou *warnings* verificados no seu código fonte. As anotaões que se encontram no exemplo são:

requires que especifica as pré-condições, condições que têm de ser verificadas imediatamente antes da execução do método, podendo já não ser válidas depois da sua execução.

ensures que especifica as pós-condições, condições que têm de ser verificadas imediatamente após a execução do método.

pure que indica que o método é puro, ou seja, não altera o valor das variáveis.

modifies que indica que o método modifica o valor de uma variável específica.

invariant que representa as invariantes e são pré e pós-condições de todos os métodos. Podem não ser mantidas durante a execução de um método, mas têm que ser asseguradas antes e depois da sua execução.

Suporte para Concorrência Para suportar controlo de concorrência (detecção de *race conditions* e/ou *deadlocks*), é utilizada a anotação `monitored L` para indicar que determinada variável é partilhada e que deve ser monitorizada pelo *lock* ou conjunto de *locks* em `L` [RLNS00]. Uma *thread* só adquire o direito a alterar a variável partilhada quando tem acesso à região crítica controlada por `L`.

Apesar de conseguir fazer verificação de *locks*, o ESCJava não tem suporte para as primitivas *wait* e *notify*.

2.3.3 *Typestates*

Typestates é uma abordagem que associa aos objectos uma máquina de estados que indica em cada estado, quais os métodos disponíveis e qual o estado seguinte do objecto após a sua execução/transição [SY86]. Enquanto os tipos dos objectos nos indicam todas as operações que poderemos vir a executar para um objecto, os *typestates* são utilizados para criar uma abstracção das operações disponíveis num objecto em cada momento da execução de um programa.

O paradigma orientado por objectos pode ser estendido de maneira a que cada objecto seja modelado em termos de classes e do conjunto de estados em que "está" ou poderá vir a "estar".

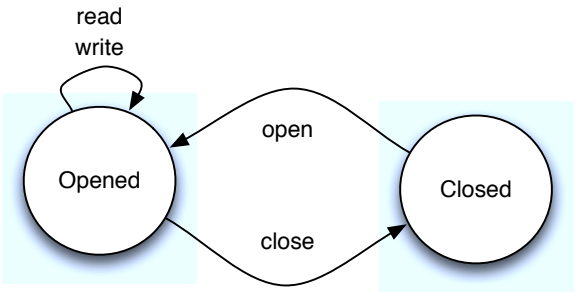

Figura 2.2: Máquina de estados da classe ficheiro

Cada objecto pode "estar" num determinado estado e, através da execução de um método, transitar para outro estado diferente. Esta transição implica a existência de um mecanismo que controle o estado dos objectos – uma máquina de estados – garantindo a integridade do sistema, permitindo que apenas sejam executadas operações disponíveis no estado "actual" de um objecto.

Um exemplo simples da máquina de estados de um objecto, neste caso um ficheiro, pode ser observada na figura 2.2. Abstraindo a maior parte das operações que podemos realizar sobre um ficheiro, podemos considerar que um ficheiro pode estar em apenas dois estados: aberto (`Opened`) e fechado (`Closed`). Quando um ficheiro está fechado, a única operação disponível é a de o abrir (`open`), transitando para o estado aberto (`Opened`). Já quando um ficheiro está aberto (`Opened`), podemos realizar operações de leitura (`read`) e escrita (`write`) ou fechar (`close`) o ficheiro, transitando novamente para o estado fechado (`Closed`).

Ao integrar estados directamente na linguagem, retiramos a complexidade de ter vários sistemas a verificar diferentes propriedades. Pode ser feita uma verificação de tipos e estados em simultâneo, retirando a necessidade de verificações separadas, que levariam a um aumento do peso dos sistemas de verificação.

Partindo do princípio que um objecto não mantém as mesmas necessidades em cada estado em que se encontra, podemos excluir dados desnecessários após a transição de um estado `A`, em que o objecto tinha determinadas necessidades, para um estado `B`, em que deixou de as ter.

Aldrich, propõe um sistema (*Plaid*) que também define uma relação de *subtyping* e coerência para o *typestate* dos objectos [ASSS09]. Esse mesmo sistema foi também estudado por Militão [MAC10], com uma proposta semelhante.

No sistema *Plaid*, tal como acontece com os tipos em Java, estados podem estender outras estados e herdar todas as operações e atributos que fazem parte de um estado pai. Um novo estado pode transitar para qualquer outro estado que seja subtipo do seu estado "pai", estendendo em certa parte a noção de herança.

Na listagem 2.6 é mostrado um exemplo prático da utilização da linguagem Plaid para gestão de um ficheiro. É definida um objecto (estado) base "pai" – `File` – que, neste caso, apenas guarda a informação sobre o nome do ficheiro. A partir deste objecto base para gestão dos ficheiros, são criados outros dois – `OpenFile` e `ClosedFile` – para poder gerir a abertura e

```
state File {
  public final String filename;
}
state OpenFile extends File {
  private CFilePtr filePtr;
  public int read() { ... }
  public void close() [OpenFile>>ClosedFile]
    { ... }
}
state ClosedFile extends File {
  public void open() [ClosedFile>>OpenFile]
    { ... }
}
```

Listagem 2.6: Exemplo de estados e hierarquias em Plaid

fecho dos ficheiros.

Tanto `OpenFile` como `ClosedFile` estendem `File`, herdando deste o campo `filename`. No caso de `OpenFile` é ainda incluido o campo de apontador para o ficheiro, a operação de leitura e fecho e, no caso de `ClosedFile`, a operação de abertura do ficheiro. Até este ponto, é em tudo semelhante à herança de ficheiros em Java.

A extensão surge a partir da definição da transição de estados para as operações. Os estados podem estender estados e herdar todas as operações e atributos que fazem parte de um estado pai, tal como em Java, com a particularidade de um novo estado poder transitar para qualquer estado que seja subtipo do seu estado pai. Neste exemplo, um objecto no estado `OpenFile` transita para o estado `ClosedFile`, após ser executada a operação `close`, e pode voltar a transitar para o estado `OpenFile` se for executada a operação `open`. Um objecto pode também manter-se no mesmo estado depois de executar uma operação se não houver indicação de transição (p. ex. `read`).

Utilizando este tipo de anotações no código pode ser mais facilmente assegurado que, ao chamar uma função, o objecto transite para um estado correcto, ou seja, esteja de acordo com o tipo de estado que é esperado no final. Um objecto pode mudar de estado no meio do código de uma função para um estado que não corresponda ao estado final, desde que o estado final seja o correcto.

São permitidas verificações sobre o estado actual de um objecto à semelhança da comparação por `instanceof` em Java, utilizando o operador `instate`:

```
if (file instate OpenFile){ read(); (...) }
```

Aliasing e Gestão de Permissões Em [ASSS09] é também abordado o problema da gestão permissões para os objectos, sendo definidos alguns conceitos para permitir um melhor controlo de *aliasing*. Na linguagem Plaid, são reservadas algumas palavras para indicar o tipo e estado de um objecto, assim como permissões da partilha dos mesmos [BBA08]:

```
public void write (unique/shared OpenFile of){ (...) }
public void read (pure OpenFile f){ (...) }
```

- `unique`: é a única referência do objecto. Apenas um *thread* de cada vez pode aceder a este objecto, passando (ou não) a permissão para o mesmo de forma linear para outtro *thread*.
- `full`: acesso exclusivo de leitura/escrita para um qualquer número de referências. Apenas um *thread* pode alterar o objecto, mas pode haver vários *threads* a ler. O *thread* com `full permission` pode basear-se no facto de mais nenhum *thread* poder modificar o estado do objecto.
- `immutable`: podem exister várias referências para o mesmo objecto mas não podem alterar o mesmo. Todos os *threads* podem basear-se no facto de este objecto nunca alterar o seu estado.
- `pure`: dá permissões de leitura aos objectos que, no entanto, podem ser modificados por outras referências. A não ser que se esteja dentro de um bloco de instruções atómico, os *threads* não têm garantia de que o objecto mantenha o seu estado.
- `shared`: podem existir várias referências com capacidade para ler e escrever sobre o objecto, assim como ser alterados por outras referências no sistema. É a permissão menos restritiva e utilizada por defeito na linguagem Java. Um *thread* com acesso a um objecto `shared` pode modificar o seu estado e, à semelhança de um objecto `pure`, não tem garantias da manutenção do estado do objecto, a não ser que esteja dentro de um bloco de intruções atómico.

Para efectuar o controlo de concorrência todas as operações sobre variáveis, cujo acesso para modificação apenas possa ser feito por um único *thread* de cada vez, terão que ser operações atómicas. Assim, garante-se que durante aquela operação apenas um *thread* alterou o objecto. No que diz respeito às variáveis partilhadas, é proposto um sistema de transacções a partir do qual se pode voltar ao estado anterior no início da transacção corrente, caso haja um conflito durante a modificação da variável partilhada.

Para um ficheiro aberto, no processo de escrita, podemos desejar que apenas um *thread* tenha acesso ao ficheiro – `unique` – ou que vários *threads* o possam ler e modificar em simultâneo – `shared`. Já para uma operação de leitura é desejável que todos os *threads* possam ler o ficheiro em simultâneo e, no limite, ser alterados por um processo que possa, por exemplo, fechar o ficheiro – `pure`.

2.3.4 *Ownership*, Tipos e Efeitos

Através dos chamados *ownership types*, Drossopoulou *et.al* propuseram um sistema de tipos e efeitos com suporte para verificação de *ownership* e gestão de *aliasing*, que permite controlar e disciplinar o acesso a variáveis partilhadas [CD02, SD03, CDNS07].

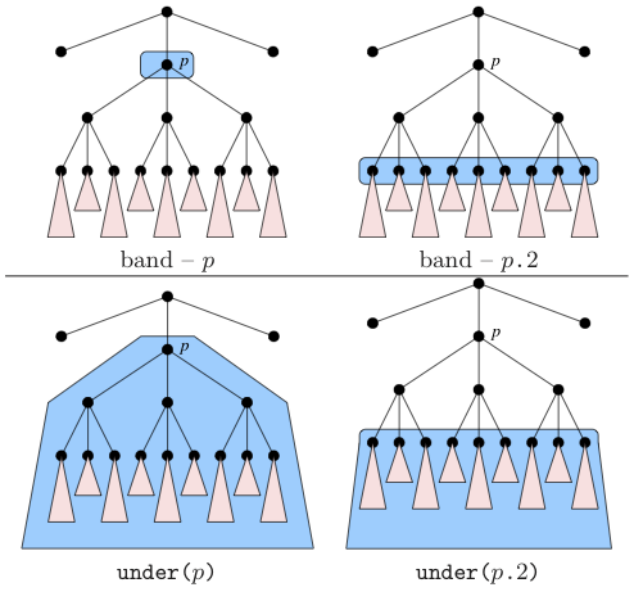

Figura 2.3: Níveis de acesso aos objectos

O controlo de acesso a objectos numa linguagem de programação orientada por objectos pode ser feita utilizando a noção de *ownership*. Uma solução para controlar o acesso aos objectos é a definição de um *owner* de um objecto a partir de outro objecto cuja representação é a dos objectos que controla. Com a introdução da noção de *ownership*, podem também ser definidos sistemas com capacidade para deduzir algumas propriedades dos programas: tipos disjuntos e efeitos disjuntos. Com estas propriedades podemos determinar propriedades de *aliasing* e interferências entre expressões concorrentes.

Num sistema de tipos orientado pela noção de *ownership*, todos os objectos têm um *owner*. Esse *owner* pode no limite ser o "mundo" – *world* – um objecto especial que controla todos os outros objectos. Na listagem 2.7, pode verificar-se uma proposta [CD02] de uma linguagem que recorre a um sistema com suporte para *ownership*. O acesso aos objectos é conseguido através da definição de níveis de permissão para cada tipo de *owner*, tal como ilustrado nas figuras 2.3 e 2.4 (retiradas de [CD02]).

O *ownership* de objectos pode também auxiliar na verificação da existência de aliases. Se na classe `Main` fossem apresentadas duas variáveis: `List<this,world> shared` e `List<this,this> encaps`, poder-se-ia concluir que as listas `shared` e `encaps` nunca seriam *aliases* uma da outra. Enquanto a variável `shared` era controlada pela classe `Main` mas os seus objectos poderiam ser controlados por qualquer classe/objecto, já a variável `encaps` apenas permitia à lista criar e operar sobre os seus próprios objectos.

Utilizando este sistema, é provado que os objectos só podem alterar o estado de outros objectos se os seus métodos tiverem acesso a esses objectos [SD03]. Assim, para um objecto `A` alterar um estado de um objecto `B`, `A` tem que ser dono de `B` (ou ter algum objecto que seja dono de `B`) e o método ter permissão para alterar `B` (ou existir um objecto dentro da árvore que tenha

```
class List<owner,data> {
  Link<this,data> head;
  void add(Data<data> d) writes under(this) {
    head = new Link<this,data>(d, head);
  }
}
class Main<> {
  List<this,world> list;
  Main() writes this {
    list = new List<this,world>;
  }
  void populate() writes under(this.1) {
    list.add(new Data<world>);
    list.add(new Data<world>);
  }
  static void main() writes under(world) {
    Main<> main = new Main<>;
    main.populate();
  }
}
```

Listagem 2.7: Exemplo de um sistema com suporte para *ownership types*

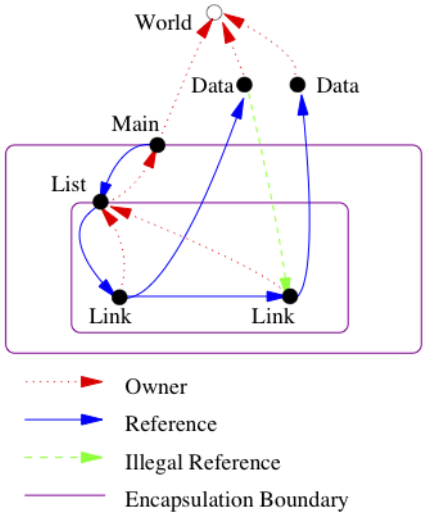

Figura 2.4: Relações de *ownership* relativas ao exemplo 2.7

```
class CombinedAccount<readonly> {
  LockLevel savingsLevel = new;
  LockLevel checkingLevel < savingsLevel;
  final Account<self:savingsLevel> savingsAccount = new Account;
  final Account<self:checkingLevel> checkingAccount = new Account;
  (...)
}
```

Listagem 2.8: Conta bancária com vários níveis de *lock*

permissão para alterar `B`).

Controlo de Concorrência através de Ownership Seguindo do sistema com suporte para *ownership* de objectos definido anteriormente, pode garantir-se que um programa com *threads* concorrentes não irá provocar *race conditions* desde que cada *thread* garanta o acesso exclusivo à raiz da árvore de controlo de um objecto.

Restringindo o acesso aos objectos pode verificar-se se cada uma das *threads* está ou não a aceder a recursos que não controla, ou não pode vir a controlar, garantindo que não irá haver interferência entre *threads*. A um nível mais avançado, os *ownership types* podem fazer verificação do controlo de concorrência através de abstracções de níveis de *locks* introduzidas na linguagem. Se cada *thread* adquirir o controlo de acesso para cada nível de *lock* por ordem decrescente, é provado que não irá haver conflito entre *threads* [BLR02].

No exemplo 2.8 (retirado de [BLR02]), uma *thread* que quisesse fazer alterações à conta bancária combinada, teria primeiro que adquirir a permissão ao nível da conta de poupança – `savingsLevel` *lock* – pois este era o nível de *lock* mais elevado. De seguida, teria que adquirir a permissão ao nível da conta à ordem – `checkingLevel` *lock* – e só aí poderia efectuar operações sobre a conta bancária partilhada.

Multiple Ownership O modelo em árvore abordado em [CD02] consegue resolver o problema de *ownership*, mas é limitado pelo facto de todos os objectos terem sempre que ter um único "*owner*", impossibilitando que um objecto pudesse ser controlado por vários *owners*. Para resolver este problema, foi proposta uma outra técnica em que os objectos são encapsulados em caixas (`boxes`) [CDNS07], podendo essas caixas intersectar-se entre si. Com este novo modelo, possibilita-se que os objectos sejam controlados por outros objectos em simultâneo. A introdução deste sistema (MOJO [CD02]) com suporte para *multiple-ownership*, permitiu que cada objecto fosse controlado por vários outros, estendendo as capacidades da linguagem.

Imaginando uma empresa com vários projectos e trabalhadores que teriam várias tarefas associadas, pode verificar-se quão útil será utilizar noção de caixas para definição de *ownership*. Na listagem 2.9 podemos encontrar um exemplo simples da partilha de controlo sobre uma variável. Neste caso, uma tarefa (`tsk`) é simultaneamente controlada por um projecto (`prj`)

```
class Project<o> {
  TaskList<this, this> tasks;
  (...)
  void add(Task<this & ?> t) {
    tasks.add(t);
  }
}

class Worker<o> {
  TaskList<this, this> tasks;
  (...)
  void add(Task<this & ?> t) {
    tasks.add(t);
  }
}
(...)
final Project<this> prj = new Project<this>();
final Worker<this> wrk = new Worker<this>();

Task<prj & wrk> tsk = new Task<prj & wrk>();
```

Listagem 2.9: Exemplo de um sistema com suporte para *multiple-ownership* [CD02]

e um trabalhador (`wrk`). Para resolver potenciais problemas de permissões entre objectos, é utilizado um mecanismo a partir do qual podem ser definidas intersecções ou disjunções entre os objectos [CD02]. Por defeito, todos os objectos criados são objectos disjuntos mas, como no exemplo do projecto com tarefas e trabalhadores, podem existir casos em que um objecto (`tsk`) é controlado por duas caixas (`wrk` e `prj`).

2.3.5 Lógica Espacial e Tipos Comportamentais

Caires [Cai08] introduziu o desenvolvimento de uma linguagem com capacidade para verificar estaticamente controlo de concorrência com recurso a memória partilhada, ao mesmo tempo que oferece suporte para execução sequencial, paralela e de repetição. Nesse trabalho é introduzido um processo de verificação para um sistema de objectos distribuídos e concorrentes.

O objectivo do sistema de tipos introduzido é de oferecer um mecanismo de disciplina de interacções e uso de recursos para sistemas com suporte para concorrência. Para oferecer esse mecanismo foi definido um sistema em que diferentes *tarefas* são consideradas independentes quando não competem pelos mesmos *recursos*. Se as *tarefas* forem independentes, não há risco de provocar interferências no acesso aos recursos utilizados, podendo assim ser executadas em simultâneo.

Ao contrário dos modelos convencionais de programação concorrente, em que são divididas as entidades em duas classes principais – activas e passivas – os *recursos* (usualmente entidades passivas) são classificadas como qualquer outro objecto, garantindo também que estes não entram em estados ilegais.

A verificação da correcção da concorrência é feita estaticamente, com o auxílio de um sistema de tipos baseado numa lógica espacial [CC03], com construtores de tipo que caracterizam a execução sequencial e paralela de métodos de objectos. Considere-se o exemplo da definição da preparação de uma viagem:

$$\textit{Viagem} \triangleq \texttt{(voo|hotel);confirmação}$$

Para preparar uma *viagem*, começamos por ter que escolher o `voo` e o `hotel`, por uma qualquer ordem ou concorrentemente("`|`") e depois (e apenas depois), podemos confirmar ambos.

É também definida uma noção de *ownership* para os comportamentos dos objectos, através do operador "$^\circ$", para indicar que o controlo do objecto passa para o método que o usa para executar. Por exemplo, um método `usa`$(V^\circ)U$ requer que o argumento do tipo V seja controlado por esse e só por esse método. Assim que `usa`(v) seja chamado, o argumento v passa apenas a ser controlado por esse método, desconhecendo-se qual o *owner* final de v, que depende da execução do método `usa`. Por outro lado, um método sem restrições de controlo – `aluga`$(V)U$ – garante que o controlo do argumento após a execução do método se mantém inalterado, podendo, no entanto, evoluir durante a execução do método.

É este o modelo seguido nesta dissertação.

Linguagem yak A linguagem yak [Mil08], implementação derivada de [Cai08] utiliza tipos comportamentais para verificar a correcção de um programa. A sintaxe é semelhante à da linguagem Java, mas adiciona às classes protocolos de utilização, que representam a forma como as instâncias podem ser utilizadas. Esses protocolos correspondem aos tipos comportamentais dos objectos, a partir dos quais se verificam os programas.

A listagem 2.10 (retirada de [Mil08]) apresenta a definição de uma classe `Bottle`, que simula, através do seu tipo comportamental, a possível utilização para uma garrafa. O tipo comportamental é definido a partir dos nomes dos métodos da classe, indicando a ordem pela qual os métodos da classe podem ser chamados. Neste caso, a utilização da garrafa define que, inicialmente, a mesma tem que ser aberta (`open`). Depois de aberta pode beber-se várias vezes o seu conteúdo (`drink*`), até que fique vazia, altura em que deve ser reciclada (excepção `Empty:recycle`). A garrafa pode ser fechada (`close`) em qualquer altura depois de ter sido aberta ou, obrigatoriamente, depois de ter sido reciclada.

O exemplo da listagem 2.11 mostra uma utilização válida da classe `Bottle`. Agarrafa é aberta e depois tenta-se beber duas vezes 50 unidades do conteúdo da garrafa, para depois a fechar. A utilização da garrafa está correcta pois, além de se seguir a ordem definida no protocolo, é tratada a excepção para quando a garrafa está vazia.

A linguagem desenvolvida no trabalho desta dissertação (capítulo 3) baseia-se, em parte, neste trabalho. Em particular, a abordagem seguida em [Cai08] e [Mil08], que define a possibilidade de utilização dos objectos a partir dos tipos comportamentais, é uma das ideias seguidas neste trabalho.

```
class Bottle{
  usage open;(drink[Empty: recycle])*;close

  integer remaining;

  void Bottle(){
    remaining = 250;
  }

  void open(){ }

  integer drink(integer amount) throws Empty{
    remaining = remaining.subtract(amount);
    if( remaining.isLessOrEqualsThan(0) )
      throw new Empty();
    return remaining;
  }

  void close(){ }

  void recycle(){ }
  void recycle(string station){ }
}
```

Listagem 2.10: Classe Bottle definida na linguagem yak

```
class Main{

  void main(){
    Bottle#open;(drink[Empty: recycle])*;close bottle = new Bottle();
    bottle.open();
    try{
      integer drink = 2;
      while(drink){
        bottle.drink(50);
        drink = drink.subtract(1);
      }
    }
    catch(Empty error){
      bottle.recycle();
      return;
    }
    bottle.close();
  }
}
```

Listagem 2.11: Utilização da classe Bottle

A linguagem yak suporta ainda a noção de *ownership* dos objectos que enriquecem bastante as capacidades da linguagem No entanto, a linguagem yak não apresenta abstracções para execução paralela e verificação do controlo de concorrência, aspecto que será o principal foco desta dissertação.

2.4 *Safe Locking*

Uma outra abordagem relacionada com sistemas de tipos que permite verificar a correcta utilização de *threads* concorrentes prende-se com a utilização de *safe locks* [FA99].

O tipo de *locks* apresentado em [FA99] consegue, se não na totalidade, em grande parte, garantir a ausência de *race-conditions* através de determinadas regras que permitem a verificação estática da sua ausência. A introdução de *locks* para efectuar controlo de concorrência dinamicamente introduz o problema da possível existência de *dead locks*. Para resolver esse problema é usado o conceito de *singleton locks (types)* que associam a cada *lock* um *singleton* para determinada região crítica de memória. Introduzindo os *locks* ao nível da verificação de tipos dos objectos a partir dos *singletons*, pode prever-se a presença de *dead locks* e assim garantir a ausência de *race conditions* nos programas.

Em particular, no trabalho apresentado em [FA99], é também definido o conceito de "permissão" que engloba um conjunto de *singleton locks*. Na verificação de um programa, todas as expressões são analisadas no contexto de uma permissão e ao ser analisadas é necessário garantir que a *thread* em que irá ser executada essa expressão tem todos os *locks* da permissão dessa expressão.

Existem algumas linguagens que introduzem o princípio de *safe lock*, como a apresentada em [VM06], que inclui primitivas para criar, adquirir e libertar *locks* e que permite criar programas concorrentes em que se garante que se os programas estiverem bem tipificados, estes não vão ficar "presos" nem vão ocorrer *race conditions*.

2.5 Inferência de Tipos

As linguagens de programação usam frequentemente anotações de tipos para adicionar informação relevante sobre as variáveis e/ou objectos de um programa. No entanto, algumas linguagens, tais como o Haskell [HHPJW96], o ML [Lei83] ou o Scala [OSV08], conseguem deduzir as informações de tipo necessárias para verificar a correcta tipificação de um programa, sem recorrer a anotações de tipo, poupando trabalho ao programador. Estas linguagens, que usam mecanismos de inferência de tipos, conseguem prever possíveis erros de execução em tempo de compilação derivados de expressões mal tipificadas, não permitindo que o programador use tipos incorrectos. Usando esta técnica de tipificação pode até, em alguns casos, retirar-se a necessidade de indicar os tipos que se estão a utilizar [Pir09].

A inferência de tipos começou a ser utilizada nos finais da década de 70, quando foi proposto um algoritmo de inferência de tipos [Mil78], conhecido com o algoritmo de Damas-Milner, tendo sido provada a sua correcção e completude [DM82].

Mais recentemente foram desenvolvidos algoritmos que combinam a inferência de tipos com *subtyping* [LW94] e que serão usados neste trabalho.

Algoritmo Damas-Milner O Algoritmo de Damas-Milner parte do conjunto de restrições de um programa, criando uma relação de restrições, que podem ser vistas como um conjunto de equações matemáticas. Essas equações são então substituídas pelos seus valores (tipos) e se em algum ponto do algoritmo não se conseguir encontrar uma substituição que a satisfaça, então quer dizer que o programa está mal tipificado. Se o algoritmo termina então quer dizer que a unificação ocorreu com sucesso e o programa está bem tipificado.

As substituições mencionadas no algoritmo de Damas-Milner, são efectuadas através da unificação de tipos [Mil78]. Um unificador é definido como sendo uma substituição que torne dois termos idênticos. Dados dois termos, t_1 e t_2, o algoritmo de unificação tenta unificá-los, ou seja, tenta encontrar uma substituição que os torne idênticos. Para o fazer são usadas duas pilhas onde são guardadas todas as equações (E) e as equações com as substituições já efectuadas (S).

Como descrito em [Rob65], o algoritmo de unificação recebe dois termos, t_1 e t_2 para serem unificados. Se em algum ponto o algoritmo é forçado a parar, é retornada uma falha na unificação. Caso contrário retorna um conjunto de substituições possíveis, S, para essa unificação. O algoritmo começa com uma pilha S de substiuições vazia e uma pilha E contendo a equação $t_1 = t_2$. Enquanto a pilha E não ficar vazia, vão sendo retiradas e verificadas as várias equações nela presentes. Para cada equação $Y = X$ em E:

1. Se X é uma variável que não ocorre em Y, substituir X por Y em E e em S; adicionar $X = Y$ a S

2. Se Y é uma variável que não ocorre em X, substituir Y por X em E e em S; adicionar $Y = X$ a S

3. Se X e Y forem variáveis ou constantes idênticas, não fazer nada.

4. Se X for $f(X_1, .., X_n)$ e Y for $f(Y_1, ..., Y_n)$ de uma função ou operação f e $n > 0$, empilhar $X_i = Y_i, i = 1...n$ em E

5. Caso contrário retornar uma falha

Tome-se como exemplo os seguintes termos t_1 e t_2:

$$t_1 = f(plus(a, b), g(a))$$

$$t_2 = f(plus(x, y), g(d))$$

O primeiro passo é igualar ambos os tipos, adicionando essa equação à pilha de equações, inicializando também a pilha de substituições vazia:

$$E : f(plus(a,b), g(a)) = f(plus(x,y), g(d)) \qquad S : \{\}$$

De seguida, é desempilhada uma equação (neste primeiro passo a única) de E. Como essa equação é uma aplicação da mesma função em ambos os membros, então aplica-se a regra (4) e as pilhas passam a:

$$E : plus(a,b) = plus(x,y), g(a) = g(z) \qquad S : \{\}$$

$$E : a = x, b = y, g(a) = g(z) \qquad S : \{\}$$

As próximas duas equações são tratadas pela regra (2) visto as variáveis do lado direito da igualdade não ocorrerem no lado esquerdo. Como já foi encontrada uma substituição possível, então essa substituição é adicionada a S:

$$E : g(a) = g(z) \qquad S : x = a, y = b$$

Para a próxima equação são seguidas novamente as regras (4) e (2), chegando a:

$$E : \{\} \qquad S : x = a, y = b, z = a$$

Como já não há mais equações em E e não foi detectada nenhuma falha, então o algoritmo termina e as substituições de S são consideradas válidas.

Inferência de Tipos com *Subtyping* A inferência de tipos pode ser combinada com a noção de *subtyping* para verificar a correcção de um programa [Sta88, Mit91]. Em particular, a noção de que um tipo τ pode substituir outro tipo σ (relação de *subtyping*), pode ser aplicada em algoritmos de inferência, para enriquecer as propriedades da linguagem. Por exemplo, uma vulgar bicicleta a pedal pode ser facilmente substituída por uma *Maxi Puch*[1].

```
class bike
  brand : string
  pedal()
  break()
  ...
```

A classe `bike` representa uma bicicleta, que tem uma determinada marca (`brand`) e as operações para pedalar e travar (`pedal` e `break`).

Uma *Maxi Puch* pode ser definida através da classe `maxipuch`, que estende a classe `bike`. As *Maxi Puch* podem fazer todas as operações de uma bicicleta normal (pedalar e travar) e

[1] Bicicletas a motor com pedais, produzidas nas décadas de 70 e 80, que permitem continuar a andar quando se acaba o combustível

```
class maxipuch extends bike
  engine : string
  start()
  stop()
  accelerate()
  ...
```

também têm uma marca. Além disso, adicionam um motor e as respectivas operações de ligar, desligar e acelerar (`start`, `stop`, `accelerate`).

Uma pessoa que necessite de se deslocar de um local para outro pode utilizar uma bicicleta, mas nada a impede de utilizar uma *Maxi Puch* para o mesmo propósito, com o benefício de poder utilizar um motor. Já se o objectivo era deslocar-se sem qualquer esforço, a única hipótese seria deslocar-se de *Maxi Puch*.

O mesmo acontece nos programas com a utilização das classes `bike` e `maxipuch`. Um programa que necessite de um objecto do tipo `bike` pode facilmente substituí-lo por um objecto do tipo `maxipuch`, mas o inverso não se verifica porque os objectos da classe `bike` não têm motor e as respectivas operações de `start`, `stop` e `accelerate`. Constata-se, portanto, que `maxipuch` é subtipo de `bike` (`maxipuch<:bike`).

As relações de *subtyping* substituem as unificações dos algoritmos de inferência, deixando de ser necessário que um tipo seja igual ao outro para poder unificar.

Em particular, podem ser aplicados os conceitos de covariância e contravariância na unificação de métodos através de *subtyping* [Cas95].

Tome-se o exemplo de dois métodos $m_1 : \tau \to \tau'$ e $m_2 : \sigma \to \sigma'$. Considera-se que m_1 é subtipo de m_2 ($\tau \to \tau' <: \sigma \to \sigma'$) se:

$$\sigma <: \tau \qquad \text{e} \qquad \tau' <: \sigma'$$

ou seja, m_1, com tipos de parâmetros mais restritos que m_2, consegue produzir um resultado com tantas ou maiores capacidades que o resultado de m_2.

3

Verificação Estática de Controlo de Concorrência

Neste capítulo define-se uma linguagem e respectivo algoritmo de tipificação. A linguagem definida é imperativa, orientada a objectos e permite execução paralela (concorrente). O principal desafio proposto é encontrar uma forma de disciplinar a utilização de variáveis que estejam em zonas de memória partilhada para que um programa não incorra em situações de *race conditions*.

Apresenta-se aqui um algoritmo de tipificação, que combina as técnicas de inferência de tipos (comportamentais) e *subtyping*, a partir do qual os programas são verificados. O algoritmo de inferência de tipos constrói um tipo com base na utilização que é de facto feita dos identificadores e cria as respectivas restrições que são verificadas por um algoritmo de *subtyping* onde os tipos inferidos são comparados com os tipos (protocolos de utilização) definidos. A linguagem de tipos permite a definição de protocolos de utilização para os objectos, por forma a restringir quais os métodos que podem ser chamados a cada momento da execução.

Para implementar a relação de *subtyping* é usada uma noção de simulação de transições a partir dos nomes dos métodos. A abordagem seguida para o algoritmo de tipificação foi bastante pragmática, definido-se regras simples para efectuar a verificação da correcção de um programa. O algoritmo foi validado através do protótipo implementado, usando uma bateria de testes criada para o efeito.

De seguida será apresentada a sintaxe da linguagem e o sistema de tipos. Serão focados os aspectos mais importantes do algoritmo implementado, nomeadamente o algoritmo de *subtyping* e de inferência de tipos.

3.1 Featherweight Concurrent Java

A sintaxe da linguagem baseia-se no *Featherweight Java* [IPW99], nomeadamente a estrutura dos construtores. A linguagem permite definir interfaces (tipos) para os quais são indicados métodos e protocolos de utilização. As classes implementam as interfaces e definem as variáveis de instância e os tipos das mesmas.

Define-se de seguida a gramática da linguagem para a qual se devem considerar as seguintes notações:

$I \in \mathcal{I}$ conjunto de nomes de interfaces
$C \in \mathcal{C}$ conjunto de nomes de classes
$m \in \mathcal{M}$ conjunto de nomes de métodos
$x \in \mathcal{X}$ conjunto de nomes de parâmetros, constantes e variáveis

3.1.1 Sintaxe

Um programa é definido por um conjunto de interfaces, classes e a expressão principal (**`main`**) que é o ponto de entrada. A sintaxe é apresentada na figura 3.1, onde podemos verificar a ordem de definição de cada parte de um programa. Inicialmente são definidas todas as interfaces e classes. Uma interface define o cabeçalho dos métodos e o protocolo de utilização que irá estar associado aos mesmos. As classes apresentam um construtor único que define as variáveis de instância e os seus tipos. Na classe são também definidos o corpo dos métodos e indicado qual o protocolo de utilização que irá ser usado, ligando a classe à interface através da palavra reservada **implements**. São utilizados os usuais tipos básicos de uma linguagem de programação **void**, **int**, **boolean** e **string**, além dos tipos das interfaces ou classes a partir dos seus identificadores (id).

Protocolos de Utilização Na definição das interfaces são também definidos os protocolos de utilização, que são construídos a partir dos nomes dos métodos através de um conjunto de operadores definidos para o efeito.

A construção de um protocolo de utilização é feito através da combinação de expressões de utilização em que $U;U$ representa a execução sequencial, $U|U$ a execução paralela, $U\&U$ a escolha, U^* a repetição em sequência e $U!$ a replicação, como em [Cai08].

Definição de um Método Os métodos seguem a definição habitual, à semelhança da linguagem Java. Têm um tipo de retorno (T), são identificados pelo seu nome (m) e definem também os seus parâmetros ($\overline{x:T}$). O corpo de um método é uma expressão. Sendo uma linguagem com suporte para concorrência, é também possível utilizar a anotação **sync** para indicar que um método é sincronizado, ou seja, bloqueia o acesso ao objecto a todos os outros métodos sincronizados até que o próprio método termine a sua execução.

$$
\begin{array}{lcll}
P & ::= & \overline{D}\ \textbf{main}\ \{\ e\ \} & \text{(Programa)} \\
 & & & \\
D & ::= & & \\
 & | & \textbf{interface}\ I\ \{\ \overline{T\ m\ (\overline{x : T})}\ \textbf{usage}\ \{\ U\ \}\ \} & \text{(Interface)} \\
 & | & \textbf{class}\ C\ (\overline{x : T})\ \textbf{implements}\ T\ \{\ \overline{M}\ \} & \text{(Classe)} \\
 & & & \\
T & ::= & \textbf{void} \mid \textbf{int} \mid \textbf{boolean} \mid \textbf{string} \mid I \mid C & \text{(Tipos)} \\
 & & & \\
U & ::= & & \text{(Protocolos)} \\
 & | & U\ \&\ U & \text{(Opção)} \\
 & | & U \mid U & \text{(Paralelo)} \\
 & | & U\ ;\ U & \text{(Sequência)} \\
 & | & U* & \text{(Repetição)} \\
 & | & U! & \text{(Replicação)} \\
 & | & m & \text{(Nome de um Método)} \\
 & & & \\
M & ::= & & \\
 & | & T\ m\ (\overline{x : T})\ \{\ e\ \} & \text{(Método } \textit{Standard}\text{)} \\
 & | & \textbf{sync}\ T\ m\ (\overline{x : T})\ \{\ e\ \} & \text{(Método Sincronizado)}
\end{array}
$$

Figura 3.1: Sintaxe FWCJ (definições)

Sintaxe para Expressões Todas as construções da linguagem são expressões. As expressões que poderiam ser vistas como um comando, comportam-se como tal, e para efeito do sistema de tipos devolvem **void** na sua avaliação. Na figura 3.2 apresenta-se a sintaxe das expressões da linguagem. São definidos os habituais operadores de aritmética, igualdade, comparação e condicionais. Para facilitar o programador, além da habitual composição sequencial de expressões, é adicionado o operador | que modela uma execução paralela de expressões.

Existem ainda as expressões como o **if-else** para as expressões condicionais, o **let** para uma declaração local de variáveis e a afectação para tipos básicos. A instanciação de novos objectos é feita através da palavra reservada **new** e a chamada de métodos usa a sintaxe habitual do acesso por "ponto". São também oferecidas as operações de negação para valores boleanos (**not**) e de *output* através da palavra reservada **out**. Os valores são também considerados expressões e podem ser valores básicos (números inteiros, *strings* ou boleanos – **int**, **string**, **boolean**) ou identificadores para valores básicos já definidos – x.

A implementação do interpretador para as expressões da forma $e_1 \mid e_2$ cria duas novas *threads*, uma para cada expressão, que executam independentemente uma da outra. O programa segue a sua normal execução quando as duas *threads* terminarem.

op	::=		(Operadores)
	|	+ | − | $\star$ | /	(Aritméticos)
	|	== | !=	(Igualdade)
	|	> | < | >= | <=	(Relacionais)
	|	**and** | **or**	(Condicionais)
	|	|	(Paralelo)
	|	;	(Sequência)
	|	:=	(Afectação)
e	::=		(Expressões)
	|	v	(Valor)
	|	$e\ op\ e$	(Operador Binário)
	|	**if** e **then** e **else** e	(If Else)
	|	**while** e **do** e	(Ciclo While)
	|	**let** $id\ =\ e$ **in** e	(Declaração Local)
	|	$x.m(\overline{e})$	(Chamada de Método)
	|	**not** (e)	(Negação)
	|	**out** (e)	(Output)
	|	**new** $id\ (\ \overline{e}\ )$	(Instanciação)
	|	$\{e\}$ | (e)	(Blocos de Expressões)
v	::=	s | n | **true** | **false** | **null** | x	(Valores)

Figura 3.2: Sintaxe FWCJ (expressões)

τ	::=		(Tipos Comportamentais)
		$\tau \,\&\, \tau$	(Opção)
	|	$\tau \mid \tau$	(Paralelo)
	|	$\tau \,;\, \tau$	(Sequência)
	|	$\tau*$	(Repetição)
	|	$\tau!$	(Replicação)
	|	stop	(Vazio/Tipo Terminal)
	|	γ	(Tipo Básico)
	|	$m : (\overline{\tau}) \rightarrow \tau$	(Tipo de um Método)
	|	$C : [\overline{\tau}] \Rightarrow \tau$	(Tipo de uma Classe)
	|	X	(Variável de Tipo Nova)
	|	id	(Identificador de um Tipo)
	|	**ref**(τ)	(Referência para τ)
	|	**sync**(τ)	(Tipo Sincronizado)
	|	**void**	(Void)
γ	::=	**int** | **boolean** | **string**	(Tipos Básicos)

Figura 3.3: Linguagem de Tipos

3.2 Tipificação

O sistema de tipos implementado utiliza um algoritmo de simulação para verificar a correcta tipificação de um programa. Os tipos comportamentais são associados a um algoritmo de *subtyping* e a técnicas de inferência de tipos para auxiliar o algoritmo de simulação. Antes da apresentação das regras gerais de tipificação serão explicados todos os algoritmos auxiliares ao sistema de tipos, para melhor se compreender o seu funcionamento.

Inicialmente importa esclarecer algumas convenções e notações que serão utilizadas ao longo da descrição do sistema de tipos, tal como indicadas na figura 3.3

Os tipos básicos são explicitamente distinguidos dos demais pois são os únicos que podem ser usados em afectações. Os tipos dos objectos, devido à sua complexidade, resultante da utilização de tipos comportamentais (protocolos de utilização), são um problema mais delicado que poderá ser estudado num trabalho futuro. A forma como se podem interromper e/ou transportar protocolos de utilização (tipos comportamentais) entre identificadores não é trivial, implicando o estudo mais aprofundado de noções de *aliasing* e *ownership*.

A inferência de tipos leva à necessidade de introduzir a notação para incógnitas na tipificação. Em determinados momentos da verificação de um programa o tipo de um identificador pode ser desconhecido, como tal, são utilizadas incógnitas de tipo ($X, Y, Z, \ldots$) para representar esses tipos indeterminados.

Os tipos podem ser anotados com **ref**(τ) e **sync**(τ), que auxiliam na identificação de, respectivamente, referências para outros tipos e de tipos que são utilizados em métodos sincronizados. A notação **ref**(τ) é utilizada na tipificação de identificadores que tenham sido afectados

para ser possível reconhecer possíveis interferências entre "escritas".

3.2.1 Subtyping

Define-se a relação de subtyping entre os tipos τ e σ, escrita $\tau <: \sigma$, segundo o princípio da substitutividade aplicada a objectos com protocolos de utilização [LW94, Cai08]. Neste caso, a propriedade de subtyping (definição 3.3, página 43) resume-se a "se $\tau <: \sigma$ então sempre que é possível chamar um método num objecto com o tipo σ, então deve ser possível chamar o mesmo método num objecto com o tipo τ". Para implementar esta relação define-se um sistema de transições etiquetadas para os tipos, com base nos protocolos de utilização. A relação de *subtyping* define-se através de um algoritmo de simulação neste sistema de transições.

Implementa-se a função de transição de um tipo com base na noção de frente de um tipo. Dado um tipo arbitrário τ, define-se a frente de τ ($\mathcal{F}(\tau)$) como o conjunto de métodos que podem ser executados de imediato sobre um objecto desse tipo.

Assim, um tipo τ é considerado subtipo de um tipo σ ($\tau <: \sigma$) se for capaz de efectuar todas as transições que o tipo σ é capaz e, se após as transições, a relação de *subtyping* se mantiver. Em particular, sempre que o tipo τ terminar σ também deve ser capaz de terminar. A relação de transição é descrita na definição 3.1 (pág. 40).

Ilustra-se intuitivamente o algoritmo de *subtyping* usando os tipos: $\tau \triangleq m_1|m_2$ e $\sigma \triangleq m_1; m_2$. O tipo τ descreve a chamada concorrente dos métodos m_1 e m_2, enquanto o tipo σ descreve uma provável chamada sequencial dos mesmos dois métodos. Para os tipos τ e σ as frentes são $\mathcal{F}(\tau) = \{m_1, m_2\}$ e $\mathcal{F}(\sigma) = \{m_1\}$. É de notar que nas frentes do tipo σ consta apenas o método m_1, pois o método m_2 só pode ser chamado depois de o método m_1 terminar. Já as frentes do tipo τ apresentam ambos os métodos, pois o protocolo de execução paralelo permite chamar os métodos em concorrência. Intuitivamente é razoável admitir que as seguintes transições são válidas para cada um dos tipos.

$$\tau = m_1 \mid m_2 \xrightarrow{m_1} m_2 \xrightarrow{m_2} \mathsf{stop}$$

$$\tau = m_1 \mid m_2 \xrightarrow{m_2} m_1 \xrightarrow{m_1} \mathsf{stop}$$

$$\sigma = m_1 \; ; \; m_2 \xrightarrow{m_1} m_2 \xrightarrow{m_2} \mathsf{stop}$$

Deste conjunto de transições, em que $\tau \xrightarrow{m} \tau'$ representa a chamada do método m num objecto do tipo τ e em que o tipo τ' é o tipo depois da chamada do método m, pode concluir-se que $\tau <: \sigma$ mas não o contrário.

Sistema de Transições Etiquetadas As transições possíveis para um dado tipo são feitas em relação a um nome de método, usado como etiqueta da transição. A transição de um tipo τ por um dado método m para outro tipo τ' corresponde à chamada do método m num objecto do tipo τ, e o tipo τ' corresponde ao resto do comportamento do objecto.

$$(\text{MÉTODO}) \quad m \xrightarrow{m} \mathsf{stop} \qquad (\text{SEQ}_\text{M}) \quad m \,;\, \tau \xrightarrow{m} \tau \qquad (\text{PAR}_\text{ME}) \quad m \mid \tau \xrightarrow{m} \tau \qquad (\text{PAR}_\text{MD}) \quad \tau \mid m \xrightarrow{m} \tau$$

$$(\text{ESCOLHA}_\text{ME}) \quad m \,\&\, \tau \xrightarrow{m} \mathsf{stop} \qquad (\text{ESCOLHA}_\text{MD}) \quad \tau \,\&\, m \xrightarrow{m} \mathsf{stop}$$

$$(\text{ESCOLHA}_\text{E}) \quad \frac{\tau \xrightarrow{m} \tau''}{\tau \,\&\, \tau' \xrightarrow{m} \tau''} \qquad (\text{ESCOLHA}_\text{D}) \quad \frac{\tau' \xrightarrow{m} \tau''}{\tau \,\&\, \tau' \xrightarrow{m} \tau''}$$

$$(\text{RPT}_\text{S}) \quad \frac{\tau \xrightarrow{m} \mathsf{stop}}{\tau* \xrightarrow{m} \tau*} \qquad (\text{RPL}_\text{S}) \quad \frac{\tau \xrightarrow{m} \mathsf{stop}}{\tau! \xrightarrow{m} \tau!} \qquad (\text{PAR}_\text{E}) \quad \frac{\tau \xrightarrow{m} \tau'' \quad \tau \neq m}{\tau \mid \tau' \xrightarrow{m} \tau'' \mid \tau'} \qquad (\text{PAR}_\text{D}) \quad \frac{\tau' \xrightarrow{m} \tau'' \quad \tau' \neq m}{\tau \mid \tau' \xrightarrow{m} \tau \mid \tau''}$$

$$(\text{RPT}) \quad \frac{\tau \xrightarrow{m} \tau'}{\tau^* \xrightarrow{m} \tau'; \tau^*} \qquad (\text{RPL}) \quad \frac{\tau \xrightarrow{m} \tau'}{\tau! \xrightarrow{m} \tau' | \tau!} \qquad (\text{SEQ}) \quad \frac{\tau \xrightarrow{m} \tau'' \quad \tau \neq m}{\tau \,;\, \tau' \xrightarrow{m} \tau'' \,;\, \tau'} \qquad (\text{SEQ}_\text{RPT}) \quad \frac{\tau' \xrightarrow{m} \tau''}{\tau* \,;\, \tau' \xrightarrow{m} \tau''} \qquad (\text{SEQ}_\text{RPL}) \quad \frac{\tau' \xrightarrow{m} \tau''}{\tau! \,;\, \tau' \xrightarrow{m} \tau''}$$

Figura 3.4: Regras para o Sistema de Transições Etiquetadas

Definição 3.1. Um tipo τ transita para o tipo τ' pela chamada do método m se é possível derivar $\tau \xrightarrow{m} \tau'$ através das regras da figura 3.4.

Note-se que as regras estão feitas de maneira a não aparecer o tipo terminal (stop) a não ser quando não há mais transições possíveis e assim facilitar a implementação da relação. Sempre que um método aparece sozinho ou ao nível de topo de um qualquer operador, a transição é feita pelas regras (MÉTODO), (SEQ_M), (PAR_M) e (ESCOLHA_M). O sistema de transições apresentado é não determinista pois existem tipos onde há a possibilidade de aplicar mais que uma transição em cada momento Por exemplo, se um método ocorrer várias vezes no tipo em paralelo, o algoritmo de *subtyping* que usa estas transições terá em conta todas as hipóteses de transição.

No caso de uma sequência onde o primeiro tipo não é um método, a regra (SEQ) faz a transição no tipo da esquerda, excepto no caso em que o tipo da esquerda é um protocolo de repetição (SEQ_RPT) ou replicação (SEQ_RPL). Como os protocolos de repetição e replicação podem ser "ignorados", se estiverem do lado esquerdo de uma sequência existe também a possibilidade de os "saltar" e de fazer a transição no tipo do lado direito. Por exemplo, para uma sequência $\tau*; \tau'$ transitar pelo método m temos duas possibilidades diferentes: ou por τ ou por τ'. Neste caso, o tipo resultado não inclui o tipo $\tau*$. Ao contrário da composição sequencial, a composição paralela não estabelece a ordem pela qual um tipo pode transitar. Como tal, um tipo $\tau \mid \tau'$ tanto pode transitar pelo tipo da esquerda (PAR_E) como pelo tipo da direita (PAR_D). Após a transição por τ, mantém-se a composição paralela com τ' e após a transição por τ', mantém-se a composição paralela com τ. No caso da escolha também é possível transitar pelo lado esquerdo (ESCOLHA_E) ou pelo lado direito (ESCOLHA_D). Após a transição, em que é escolhido um ramo, o outro é "esquecido". O resultado final é o tipo resultante da transição pelo ramo escolhido. Um tipo da forma $\tau*$ transita pela regra (REPETIÇÃO) e compõe, em

sequência, o tipo resultante da transição em τ com o tipo inicial $\tau*$. Um tipo da forma $\tau!$ transita por (REPLICAÇÃO) para um novo tipo paralelo em que um dos ramos corresponde à transição do tipo encapsulado pelo protocolo de replicação e o outro ramo ao próprio protocolo de replicação.

Para saber que transições existem para um dado tipo define-se a seguir a noção de frente de um tipo.

Frente de um Tipo O conjunto de métodos da frente de um tipo corresponde a todas as possibilidades de transição desse tipo, tal como apresentado na definição 3.1. A frente de um tipo ($\mathcal{F}(\tau)$) é definida indutivamente na estrutura do tipo e representa o conjunto de métodos que cada tipo pode chamar em cada momento da execução de um programa.

Definição 3.2 (Frentes)**.** As frentes de um tipo τ, escritas $\mathcal{F}(\tau)$, são definidas indutivamente na estrutura do tipo, da seguinte maneira:

$$\begin{aligned}
\mathcal{F}(\mathsf{stop}) &\triangleq \emptyset \\
\mathcal{F}(m) &\triangleq \{\{m\}\} \\
\mathcal{F}(\tau_1 \ ; \ \tau_2) &\triangleq \mathcal{F}(\tau_1) \\
\mathcal{F}(\tau_1 \mid \tau_2) &\triangleq \mathcal{F}(\tau_1) \uplus \mathcal{F}(\tau_2) \\
\mathcal{F}(\tau_1 \ \& \ \tau_2) &\triangleq \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\
\mathcal{F}(\tau*) &\triangleq \mathcal{F}(\tau) \\
\mathcal{F}(\tau!) &\triangleq \mathcal{F}(\tau)! \\
\mathcal{F}(\tau_1*; \tau_2) &\triangleq \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\
\mathcal{F}(\tau_1!; \tau_2) &\triangleq \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)
\end{aligned}$$

onde o operador $\biguplus$ representa a combinação de todas as possibilidades de união de conjuntos.

Note-se que o operador $\biguplus$ acima, no caso de ser aplicado a dois multiconjuntos contendo um único conjunto de frentes cada um, retorna o conjunto que contém a união dos dois conjuntos interiores:

$$\{\{m_1, m_2\}\} \uplus \{\{m_3, m_4\}\} = \{\{m_1, m_2, m_3, m_4\}\}$$

Se um dos operadores contiver mais do que um conjunto interior, a aplicação do operador $\biguplus$ efectua todas as combinações possíveis de união dos conjuntos:

$$\{\{m_1\}, \{m_2\}\} \uplus \{\{m_3, m_4\}\} = \{\{m_1, m_3, m_4\}\ , \{m_2, m_3, m_4\}\}$$

A aplicação do operador ! sobre um multiconjunto de frentes transforma todos os métodos de um tipo num tipo de frente especial ($m!$) que indica que o método pode ser repetido várias vezes. Fazendo esta transformação é possível reconhecer quais os métodos que se devem manter "sempre" na frente de um tipo.

$$(\{\{m_1, m_3, m_4\}\ , \{m_2, m_3, m_4\}\})! = \{\{m_1!, m_3!, m_4!\}\ , \{m_2!, m_3!, m_4!\}\}$$

Se uma sequência começar com uma repetição ou replicação, esse tipo pode ser ignorado, passando-se directamente ao lado direito da sequência. Têm-se assim dois conjuntos de frentes para este tipo de sequência: começar pelo lado esquerdo, ou ignorar o lado esquerdo e avançar apenas pelo lado direito. A frente de um tipo paralelo está associada ao operador $\biguplus$ e retorna todas as combinações possíveis da frente do lado esquerdo do operador com a frente do lado direito do operador. Pode iniciar-se a execução por qualquer um dos ramos do operador, ou até pelos dois em simultâneo.

Como a regra de transição do tipo repetição transforma o tipo encapsulado pelo operador $*$ numa sequência, a frente deste tipo corresponde à frente do tipo encapsulado. A replicação é um tipo mais complexo do que a repetição. Um tipo encapsulado por este operador poderia ser visto como estando "infinitas" vezes na frente, Usa-se a frente especial $m!$. A frente de um tipo escolha é um conjunto de conjuntos que corresponde a todas as combinações possíveis de escolha.

Considere-se o exemplo seguinte para ilustrar como seria obtida a frente de um tipo. Seja $\tau = (m_1 \;\&\; m_2) \mid (m_3 \mid m_4)$, as frentes dos dois lados do operador paralelo raíz são:

$$\mathcal{F}(m_1 \;\&\; m_2) = \{\{m_1\}, \{m_2\}\}$$

$$\mathcal{F}(m_3 \mid m_4) = \{\{m_3, m_4\}\}$$

As frentes de τ são obtidas pelas várias combinações possíveis dos dois conjuntos de conjuntos. Temos então,

$$\mathcal{F}(\tau) = \{\{m_1, m_3, m_4\}, \{m_2, m_3, m_4\}\}$$

Subtyping Depois de definidas as regras de transição e as frentes de um tipo, é possível agora definir a relação de *subtyping*.

Definição 3.3 (Relação de Subtyping)**.** Um tipo τ é subtipo de um tipo σ, escrito $\tau <: \sigma$, se $\tau = \sigma$, ou

1. Para todas as frentes $m \in \mathcal{F}(\sigma)$ e todos os tipos σ', se $\sigma \xrightarrow{m} \sigma'$, então $m \in \mathcal{F}(\tau)$ e existe o tipo τ' tal que $\tau \xrightarrow{m} \tau'$, e $\tau' <: \sigma'$, e
2. se $\tau = \mathbf{ref}(\gamma)$ então $\sigma = \gamma$ ou $\sigma = \mathbf{ref}(\gamma)$, e
3. se $\tau = \mathbf{sync}(\tau')$ então $\sigma = \tau'$ ou $\sigma = \mathbf{sync}(\tau')$, e
4. se $\tau = \tau_r \; m(\overline{x : \tau_p})$ e $\sigma = \sigma_r \; m(\overline{y : \sigma_p})$ então $\tau_r <: \sigma_r$ e $\forall_{i \in 1..n} \,.\, \sigma_{p_i} <: \tau_{p_i}$, e
5. se $\tau = C_1(\overline{x : \tau_f}) \Rightarrow \tau_u$ e $\sigma = C_2(\overline{y : \sigma_f}) \Rightarrow \sigma_u$ então $\tau_u <: \sigma_u$ e $\forall_{i \in 1..n} \,.\, \sigma_{f_i} <: \tau_{f_i}$, e
6. se $\tau = \tau'*$ ou $\tau = \tau'!$ ou $\tau = \mathsf{stop}$ então $\tau <: \mathsf{stop}$

Adicionalmente, ao item 1, verificam-se ainda que os tipos dos parâmetros e o tipo de retorno segundo o que está no item 4. Nos casos em que a verificação chega a um ciclo, considera-se

que o par está na relação, pois estes iriam continuar a simular-se indefinidamente. No item 6 τ pode ser uma forma mais complexa onde só entrem tipos da forma $\tau*$ ou $\tau!$, ou seja, que pode ser ignorados. Aplica-se também a transitividade entre relações de subtipos, ou seja se $\tau <: \tau'$ e $\tau' <: \sigma$, então $\tau <: \sigma$.

A regra geral para aplicação do algoritmo de *subtyping* indica que um tipo τ é subtipo de um tipo σ se, em qualquer estado do algoritmo, o tipo τ conseguir efectuar todas as transições que σ consegue executar. Quando a frente do supertipo for um conjunto vazio $\sigma = \mathsf{stop}$ então o subtipo também deve ser stop ou deve poder não executar mais nenhum método (item 6, $\tau <: \mathsf{stop}$). A regra geral (item 1) prevê os casos para transições em tipos comportamentais através de nomes dos métodos e é complementada com mais alguns casos para tipos específicos.

Considera-se que um tipo com um método m_1 é subtipo de um tipo com o método m_2 se as etiquetas correspondentes aos seus nomes forem iguais, o tipo de retorno do método m_1 (τ_r) for subtipo do tipo de retorno do método m_2 (σ_r) e os tipos dos parâmetros do método m_2 forem subtipo dos tipos dos pârametros do método m_1. São usadas as habituais regras de covariância e contravariância nas relações entre métodos [Car88, Cas95].

A regra para as classes funciona de forma idêntica à regra para os métodos. No entanto, não é necessário fazer qualquer comparação para as etiquetas dos nomes das classes, visto os tipos das classes serem comparados através do protocolo de utilização (interface) das mesmas.

Os tipos referência e tipos sincronizados também apresentam regras específicas, que indicam que onde é usado um tipo τ também se pode usar o mesmo tipo como referência (**ref**(τ)) ou sincronizado (**sync**(τ)).

O algoritmo de *subtyping* termina quando se chega a uma de duas situações: o supertipo (σ) já não tem mais transições possíveis ($\mathcal{F}(\sigma) = \emptyset$); ou o supertipo ($\sigma$) pode não efectuar mais nenhuma transição ($\sigma <: \mathsf{stop}$) e já foram verificadas todas as possibilidades de transição a partir desse ponto. Em qualquer dos casos, o subtipo (τ) também deve poder não efectuar mais nenhuma transição ($\tau <: \mathsf{stop}$).

Há duas (na realidade três) formas diferentes de se chegar ao caso terminal. Pode obter-se de forma directa, em que ambos os tipos são stop, ou então quando o supertipo é stop e o subtipo pode ser "ignorado". Esta segunda possibilidade acontece quando o subtipo é definido por um protocolo de repetição ($\tau*$) ou replicação ($\tau!$), que podem não efectuar qualquer transição do seu tipo interno (τ). Se o supertipo já não tem nada para executar e o subtipo também pode não executar mais nenhum método, então o algoritmo pode terminar.

3.2.2 Exemplo de Aplicação do Algoritmo de Subtyping

Para ilustrar a aplicação do algoritmo de subtyping, utiliza-se o exemplo da célula de memória apresentado no capítulo 1 (listagem 1.1, página 10). É definida uma interface `Cell` com dois métodos, `set` e `get`. Na interface é também especificado o protocolo de utilização: **`usage`**`{(set&get!)*}`. Considere-se a expressão:

```
let c = new CellImpl(0) in c.set(1);(c.get()|c.get())
```

Da expressão `c.set(1);(c.get()|c.get())` retira-se que o tipo associado ao identificador `c` tem que ser capaz de executar os métodos pela ordem que está na expressão, ou seja ser subtipo de:

$$set; (get \mid get)$$

Dado que o tipo dos objectos da classe `CellImpl` é $(set \ \& \ get!)*$, na tipificação da expressão tem que se verificar, em relação ao identificador `c`, que:

$$(set \ \& \ get!)* <: set; (get \mid get)$$

Sendo que as frentes dos dois tipos são

$$\mathcal{F}((set \ \& \ get!)*) = \{\{set, get!\}\}$$
$$\mathcal{F}(set; (get \mid get)) = \{\{set\}\}$$

pode verificar-se que o subtipo apresenta todas as possibilidades de transição do supertipo e, como tal, o algoritmo pode prosseguir. Dado que o método set é o único na frente do supertipo, as transições do subtipo por esse método são duas (pelas regras ($\text{ESCOLHA}_{\text{ME}}$) e ($\text{RPT}_{\text{S}}$)) mas o resultado é o mesmo:

$$(set \ \& \ get!)* \xrightarrow{set} (set \ \& \ get!)*$$

Se ao supertipo for aplicada a regra de transição (SEQ), obtem-se:

$$set; (get|get) \xrightarrow{set} get \mid get$$

Depois de efectuadas as transições em ambos os tipos, deve voltar a verificar-se a relação de *subtyping* para os tipos no estado seguinte:

$$(set \ \& \ get!)* <: get \mid get$$

Calculam-se novamente as frentes de cada um do tipos:

$$\mathcal{F}((set \ \& \ get!)*) = \{\{set, get!\}\}$$
$$\mathcal{F}(get \mid get) = \{\{get, get\}\}$$

A frente do subtipo apresenta o tipo $get!$, o que significa que esse método pode ser utilizado em comparações de métodos tantas vezes quantas necessárias. Neste caso, seria necessário ter uma frente do subtipo com um conjunto em que o método get aparecesse duas vezes. Como o método get está associado ao operador ! então ele poderia ser replicado para aparecer essas duas vezes e conclui-se que está de acordo que as exigências do supertipo. Como o subtipo consegue efectuar todas as transições do supertipo o algoritmo pode continuar com a simulação de transições.

O método get aparece duas vezes na frente do supertipo, logo é necessário simular as duas possibilidades de transição (por get à esquerda (PAR_{ME}) e por get à direita (PAR_{MD})). O algoritmo trata os dois casos, mas os resultados são idênticos. Para facilitar omite-se a simulação para a transição inicial pelo lado direito do operador.
Efectua-se a transição por get no subtipo utilizando as regras ($\text{ESCOLHA}_{\text{MD}}$) e ($\text{RPL}_{\text{S}}$):

$$(set \ \& \ get!)* \xrightarrow{get} get!; (set \ \& \ get!)*$$

e utiliza-se a regra (PAR_{ME}) no supertipo:

$$get|get \xrightarrow{get} get$$

chegando-se a uma nova relação de *subtyping*:

$$get!; (set \ \& \ get!)* <: get$$

Novamente, as frentes dos tipos são as seguintes:

$$\mathcal{F}(get!; (set \ \& \ get!)*) = \{\{get!\}, \{set, get!\}\}$$
$$\mathcal{F}(get) = \{\{get\}\}$$

A frente do subtipo apresenta agora dois conjuntos diferentes. A existência destes dois conjuntos deve-se à possibilidade do lado esquerdo da sequência poder não ser executada pois os tipos aos quais esteja aplicado o operador ! podem não ser executados (SEQ_{RPL}). A frente do subtipo apresenta, portanto, um conjunto para o caso em que é executada a expressão do lado esquerdo da sequência e outra para o caso em que se segue logo para a expressão do lado direito.

Como há duas combinações de transições possíveis o algoritmo divide-se em duas verificações separadas: uma para cada possibilidade de transição. São aplicadas novamente as regras da figura 3.4 e obtêm-se os seguintes casos:

$$\begin{array}{c} get!; (set \ \& \ get!)* \xrightarrow{get} get!; (set \ \& \ get!)* \\ get \xrightarrow{get} \mathsf{stop} \end{array} \quad (1) \qquad \begin{array}{c} (set \ \& \ get!)* \xrightarrow{get} get!; (set \ \& \ get!)* \\ get \xrightarrow{get} \mathsf{stop} \end{array} \quad (2)$$

O caso (1) corresponde à transição do método get no lado esquerdo da sequência e o caso (2) à mesma transição no lado direito da sequência. A nova relação de *subtyping* é já um caso terminal:

$$get!; (set \ \& \ get!)* <: \mathsf{stop}$$

Neste ponto a frente do subtipo é vazia ($\mathcal{F}(\mathsf{stop}) = \emptyset$) e aplica-se o caso 6 da definição 3.3 à relação de *subtyping*. Como o subtipo apresenta um comportamento de execução sequencial de replicações e/ou repetições, considera-se que é subtipo de stop (pode não ser executado) e o algoritmo termina, garantindo que foi feita uma utilização correcta do identificador c.

3.2.3 Inferência de Tipos

O sistema de inferência de tipos implementado não segue exactamente o tradicional algoritmo de Damas-Milner tal como descrito no capítulo 2.5. Em vez de se registarem restrições de igualdade de tipos, registam-se relações de *subtyping* que restringem o comportamento que os valores têm que ter em cada momento. Tal como no algoritmo de Damas-Milner, é atribuído um tipo "indefinido" a todas as expressões, em particular a variáveis ou chamadas de métodos, é-lhes atribuída uma variável de tipo, X. A principal diferenças para o algoritmo base surge quando se pretende fazer a "unificação" de dois tipos. Em vez de utilizar o conceito de unificação, o sistema implementado utiliza o conceito de subtipo (e supertipo) para verificar se uma variável está bem tipificada [Sta88, Mit91].

Cada variável de tipo X guarda todos os tipos dos quais é (tem que ser) subtipo e dos quais é (tem que ser) supertipo. O processo de tipificação só avança se todas as relações de todas as incógnitas forem consistentes. Assim, uma relação de subtipo $X <: \tau$ considera-se coerente se, para qualquer $\tau_s \in \mathcal{T}_{sub}(X)$, o conjunto dos subtipos de X, temos que $\tau_s <: \tau$ é verificável. O simétrico também é válido, ou seja, $\tau <: X$ se para qualquer $\tau_S \in \mathcal{T}_{super}(X)$, $\tau <: X$.

Durante a verificação de um programa é mantido um conjunto de restrições (coerentes) para as variáveis de tipo. Em particular, sempre que existe uma comparação de *subtyping* para uma variável de tipo e essa comparação é considerada válida, a relação de *subtyping* é adicionada ao conjunto de restrições.

Tome-se o exemplo da listagem 3.1, correspondente à definição de uma lista de inteiros de forma funcional, em que cada operação de adição (`add`) retorna uma nova lista de inteiros com o novo elemento. Pode verificar-se, através da definição da classe `IntListImpl`, que o método `add` devolve uma nova instância dessa mesma classe, do tipo `IntList`.

A expressão de entrada do programa (**`main`**), define a criação recursiva de uma lista com 3 inteiros: {1,2,3}. A verificação da correcção dessa expressão é feita através da criação de um conjunto de restrições.

Simplificando, numa análise *bottom-up* ao código da expressão de entrada, retiram-se os seguintes tipos e restrições de cada uma das expressões internas:

$$\Delta_{out} = \{list3 : get \to X\}\ , C_{out} = \emptyset \quad \vdash \texttt{out(list3.get(3))}$$

$$\Delta_3 = \{list2 : add \to Y\}\ , C_3 = \{Y <: get \to X\} \quad \vdash \texttt{let list3 = list2.add(3) in}$$

$$\Delta_2 = \{list1 : add \to Z\}\ , C_2 = \left\{ \begin{array}{l} Y <: get \to X, \\ Z <: add \to Y \end{array} \right\} \quad \vdash \texttt{let list2 = list1.add(2) in}$$

$$\Delta_1 = \emptyset\ , C_1 = \left\{ \begin{array}{l} Y <: get \to X, \\ Z <: add \to Y, \\ IntList <: add \to Z \end{array} \right\} \vdash \texttt{let list1 = new IntListImpl(1,null) in}$$

Até à análise da declaração inicial (**`let`** `list1 =` **`new`** `IntListImpl(1,null)` **`in`** `...`), o conjunto de restrições propagado (C_2) apenas adiciona as várias restrições ao conjunto porque não tem possibilidades de comparar com nenhum tipo concreto.

```
interface IntList {
  int get(pos:int);
  IntList add (n:int);
  IntList remove(pos:int);

  usage{ (get!&add&remove)* }
}

class IntListImpl(head:int, tail:IntList) implements IntList {
  int get(pos:int) { ... }
  IntList remove (pos:int) { ... }
  IntList add (n:int) {
    if (tail == null) then { let newTail = new IntListImpl(n,null) in
      new IntListImpl(head,newTail) } }
    else{ let newTail = tail.add(n) in new IntListImpl(head,newTail) }
  }
}

main {
  let list1 = new IntListImpl(1,null) in
  let list2 = list1.add(2) in
  let list3 = list2.add(3) in
    out(list3.get(3))
}
```

Listagem 3.1: Lista de Inteiros (funcional)

O último conjunto de restrições, C_1, já inclui uma comparação com um tipo concreto, $IntList$, derivado da criação de um objecto da classe `IntListImpl`.

Integrando esta restrições no algoritmo de simulação, os tipos vão sendo propagados e verificados ao longo das restrições (para facilitar substituem-se as variáveis de tipo pelos tipos concretos):

$$
\begin{array}{lll}
\textbf{IntList} <: add \to Z & , Z <: add \to Y & , Y <: get \to X \\
IntList <: add \to \textbf{IntList} & , Z <: add \to Y & , Y <: get \to X \\
IntList <: add \to IntList & , \textbf{IntList} <: add \to Y & , Y <: get \to X \\
IntList <: add \to IntList & , IntList <: add \to \textbf{IntList} & , Y <: get \to X \\
IntList <: add \to IntList & , IntList <: add \to IntList & , \textbf{IntList} <: get \to X \\
IntList <: add \to IntList & , IntList <: add \to IntList & , IntList <: get \to \textbf{int}
\end{array}
$$

O algoritmo inicia a execução a partir da relação $IntList <: add \to Z$ e reconhece que o tipo $IntList$ tem uma transição pelo método add. Nesse sentido, comparam-se os tipos dos dois métodos, os tipos pré-definidos e os tipos inferidos. Ao comparar os dois métodos verifica-se que no tipo $IntList$ o método add retorna o tipo $IntList$. Assim, deve adicionar-se uma nova restrição de *subtyping* para relacionar os dois tipos de retorno: $IntList <: Z$. Para adicionar essa nova restrição, deve verificar-se que o tipo $IntList$ é subtipo de todos os supertipos de Z. Das restrições propagadas encontra-se a relação $Z <: add \to Y$ e, portanto deve verificar-se que

$IntList <: add \rightarrow Y$. O algoritmo continua e repete o mesmo processo tantas vezes quantas necessárias até verificar que todas as restrições são válidas (ou encontrar uma relação inválida). Para este exemplo, como todas as restrições são válidas, o programa está bem tipificado.

3.2.4 Algoritmo de Tipificação

Define-se agora o algoritmo de tipificação, através de uma análise de casos.

Definição 3.4 (Tipificação de uma Expressão)**.** Uma expressão e está bem tipificada em relação a um conjunto coerente de restrições C, com o tipo τ, ambiente Δ e restrições resultantes C' se

$$\mathsf{check}(e, C) = (\tau, \Delta, C') \wedge C' \text{ é coerente (}\textit{sound}\text{)}$$

de acordo com os casos das figuras 3.7, 3.8 e 3.9.

A tipificação de uma expressão (e) é feita com relação a um conjunto de restrições C de *subtyping* para as variáveis de tipo (ver secção 3.2.3, página 47) e devolve um triplo (τ, Δ, C') que representa, respectivamente, o tipo da expressão e, o ambiente, ou seja a utilização que é feita dos identificadores livres de e, e um conjunto (propagado) de restrições que resulta da tipificação da expressão e.

Note-se que o conjunto de restrições C representa as relações de *subtyping* para variáveis de tipo, obtidas por inferência na tipificação das expressões e que vão sendo propagadas entre os vários processos de tipificação, devendo ser coerentes no final da tipificação.

O ambiente Δ guarda os tipos inferidos para identificadores livres e, à semelhança do conjunto de restrições, vai sendo alterado para incluir as associações do forma $id : \tau$ para todos os identificadores encontrados na tipificação de uma expressão. Os ambientes são ainda peça fundamental na verificação de interferências indesejadas em programas concorrentes, através das suas opções de junção que verificam a correcta utilização dos identificadores nas expressões. As operações de junção para ambientes são utilizadas para combinar dois ambientes conforme as expressões a que estão associados. Por exemplo, a verificação de uma expressão do tipo $e_1; e_2$, deriva da verificação intermédia de cada uma das expressões em separado. Cada uma dessas verificações devolve um ambiente, respectivamente Δ_1 para e_1 e Δ_2 para e_2. Esses dois ambientes devem depois ser combinados $(\Delta_1; \Delta_2)$ para corresponderem ao ambiente final da expressão $e_1; e_2$.

3.2.4.1 Regras e Construções Auxiliares

Na definição do algoritmo de tipificação são utilizadas algumas regras e construções auxiliares, nomeadamente a transformação de protocolos em tipos e a junção de ambientes, que são apresentadas de seguida.

Os tipos comportamentais utilizados na linguagem são definidos à custa de protocolos de utilização. Esses tipos são definidos nas interfaces, identificados pelos nomes dos métodos e

relacionados pelos diversos operadores de protocolos disponíveis. A verificação das classes fica, portanto, dependente da utilização dos campos da classe que é feita por cada um dos métodos. Os métodos podem ser chamados zero, uma ou mais vezes consoante o protocolo de utilização definido pela interface. Como tal, os ambientes resultantes da sua verificação devem ser agregados segundo esse mesmo protocolo. Foi então utilizada uma função auxiliar que associa protocolos a pares (tipo, ambiente) a partir de um protocolo de uma classe, dos tipos e ambientes (que ditam a utilização dos campos) resultantes de tipificar cada método.

$$\mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau, \Delta)$$

em que I representa a interface do protocolo utilizado e $(\overline{\tau_{m_i}, \Delta_i})$ os pares (tipo, ambiente) a serem transformados. O resultado será um único par (τ, Δ) – (tipo, ambiente) – depois de agregados os tipos e ambientes iniciais. As construções são simples, bastando aplicar a cada par o operador correspondente ao protocolo de utilização.

A função usageToType está definida inductivemente na estrutura do protocolo por análise de casos na figura 3.5. A regra base para a construção é a regra para um método que retorna o par que corresponde a esse mesmo método.

$$\mathsf{usageToType}(m_i, (\overline{\tau_m, \Delta})) = (\tau_{m_i}, \Delta_{m_i})$$

A sequência $(I_1; I_2)$ é agregada através de sequência de tipos resultado e ambientes.

$$\begin{array}{l} \text{if } I = (I_1; I_2) \\ \quad \text{and:} \\ \quad\quad \mathsf{usageToType}(I_1, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1, \Delta_1) \\ \quad\quad \mathsf{usageToType}(I_2, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_2, \Delta_2) \\ \mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1; \tau_2, \Delta_1; \Delta_2) \end{array}$$

Os dois pares resultado obtidos para I_1 e I_2 são combinados com o operador sequência retornando o par $(\tau_1; \tau_2, \Delta_1; \Delta_2)$. A combinação de ambientes através dos operadores de tipos é descrita a seguir nesta secção. As restantes construções são definidas de forma análoga.

Ambientes de Tipificação Os ambientes de tipificação têm particular importância na verificação da correcta tipificação de um programa. Através dos ambientes podem ser feitas algumas verificações nas operações de agregação, garantindo a ausência de interferências indesejadas nos acessos a memória partilhada do programa (figura 3.6).

Define-se um ambiente de tipificação pela gramática

$$\Delta ::= \emptyset \mid x : \tau, \Delta$$

É usada a notação $\Delta(x)$ para denotar o tipo associado ao identificador x no ambiente Δ. Se esse identificador x existir no ambiente é retornado o respectivo tipo τ, caso contrário, é indicado

$$\mathsf{usageToType}(m_i, (\overline{\tau_m, \Delta})) = (\tau_{m_i}, \Delta_{m_i})$$

if $I = (I_1; I_2)$
  and:
    $\mathsf{usageToType}(I_1, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1, \Delta_1)$
    $\mathsf{usageToType}(I_2, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_2, \Delta_2)$
$\mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1; \tau_2, \Delta_1; \Delta_2)$

if $I = (I_1 | I_2)$
  and:
    $\mathsf{usageToType}(I_1, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1, \Delta_1)$
    $\mathsf{usageToType}(I_2, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_2, \Delta_2)$
$\mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1 | \tau_2, \Delta_1 | \Delta_2)$

if $I = (I_1 \& I_2)$
  and:
    $\mathsf{usageToType}(I_1, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1, \Delta_1)$
    $\mathsf{usageToType}(I_2, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_2, \Delta_2)$
$\mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau_1 \& \tau_2, \Delta_1 \& \Delta_2)$

if $I = (I'*)$
  and:
    $\mathsf{usageToType}(I', (\overline{\tau_{m_i}, \Delta_i})) = (\tau, \Delta)$
    $\Delta' = \{\overline{x : \tau_x *}\} \quad \forall_{x:\tau_x} \in \Delta$
$\mathsf{usageToType}(I, (\overline{\tau_{m_i}, \Delta_i})) = (\tau *, \Delta')$

if $I = (I'!)$
  and:
    $\mathsf{usageToType}(I', (\overline{\tau_{m_i}, \Delta_i})) = (\tau, \Delta)$
    $\Delta' = \{\overline{x : \tau_x !}\} \quad \forall_{x:\tau_x} \in \Delta$
$\mathsf{usageToType}(I!, (\overline{\tau_{m_i}, \Delta_i})) = (\tau !, \Delta')$

Figura 3.5: Agregação de tipos e ambientes segundo um protocolo.

$$\Delta \,;\, \emptyset = \Delta \qquad \emptyset \,;\, \Delta = \Delta$$
$$\Delta \mid \emptyset = \Delta \qquad \emptyset \mid \Delta = \Delta$$
$$\Delta \,\&\, \emptyset = \Delta \qquad \emptyset \,\&\, \Delta = \Delta$$

$$\Delta* = \bigcup_{x \in D} x : \tau* \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array}$$

$$\mathbf{sync}(\Delta) = \bigcup_{x \in D} x : \mathbf{sync}(\tau) \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array}$$

$$\Delta_1 \,;\, \Delta_2 = \bigcup_{x \in D} x : (\tau_1 \,;\, \tau_2) \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array}$$

$$\Delta_1 \,\&\, \Delta_2 = \bigcup_{x \in D} x : (\tau_1 \,\&\, \tau_2) \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array}$$

$$\mathsf{hasRefs}(\tau) = \begin{cases} \mathbf{true}, & \text{if } \tau = \mathbf{ref}(\tau') \\ \mathsf{hasRefs}(\tau_1) \vee \mathsf{hasRefs}(\tau_2), & \text{if } \tau = \tau_1 \; op \; \tau_2,\; op \in \{ \,;\, ,\, \mid ,\, \& \,\} \end{cases}$$

$$\mathsf{isSync}(\tau) = \begin{cases} \mathbf{true}, & \text{if } \tau = \mathbf{sync}(\tau') \\ \mathsf{isSync}(\tau_1) \wedge \mathsf{isSync}(\tau_2), & \text{if } \tau = \tau_1 \; op \; \tau_2,\; op \in \{ \,;\, ,\, \mid ,\, \& \,\} \end{cases}$$

$$\Delta! = \bigcup_{x \in D} x : \tau! \text{ , if } \neg\mathsf{hasRefs}(\tau) \;\vee\; \mathsf{isSync}(\tau) \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array}$$

$$\Delta_1 \mid \Delta_2 = \bigcup_{x \in D} x : \tau \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array}$$
$$\tau = (\tau_1 \mid \tau_2), \text{ if } \mathsf{checkParallel}(\tau_1, \tau_2) \;\vee\; (\tau_1 = \mathbf{void}) \;\vee\; (\tau_2 = \mathbf{void})$$
$$\mathsf{checkParallel}(\tau_1, \tau_2) = (\neg\mathsf{hasRefs}(\tau_1) \wedge \neg\mathsf{hasRefs}(\tau_2)) \;\vee\; (\mathsf{isSync}(\tau_1) \wedge \mathsf{isSync}(\tau_2))$$

Figura 3.6: Junção de ambientes

que o identificador não existe no ambiente. Se na agregação de dois ambientes Δ_1 e Δ_2, para o mesmo identificador x, é indicado que um dos ambientes não tem qualquer referência para x (por exemplo, $x \notin \Delta_1$ e $x \in \Delta_2$), a agregação corresponde à associação desse identificador no ambiente que o contém ($x : \Delta_2(x)$).

A agregação de um ambiente com um ambiente vazio retorna o próprio ambiente.

$$\begin{array}{rr} \Delta \,;\, \emptyset = \Delta & \emptyset \,;\, \Delta = \Delta \\ \Delta \mid \emptyset = \Delta & \emptyset \mid \Delta = \Delta \\ \Delta \,\&\, \emptyset = \Delta & \emptyset \,\&\, \Delta = \Delta \end{array} \qquad \text{(Ambientes Vazios)}$$

Os operadores de repetição e sincronização limitam-se a encapsular cada um dos tipos guardados no ambiente num novo tipo, correspondente à aplicação do operador ao tipo guardado. Por exemplo, para um ambiente Δ que contenha uma única associação, $x : \tau$, a aplicação do operador $*$ iria retornar um novo ambiente $\Delta*$ com a associação $x : \tau*$. O comportamento do operador **sync** é análogo ao operador $(*)$.

$$\Delta* = \bigcup_{x \in D} x : \tau* \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array} \qquad \text{(Repetição)}$$

$$\textbf{sync}(\Delta) = \bigcup_{x \in D} x : \textbf{sync}(\tau) \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array} \qquad \text{(Sincronização)}$$

Importa também esclarecer que os tipos básicos podem (e devem) ser transformados através da junção de ambientes. Pode, por exemplo, ter-se um ambiente Δ com a associação x : **int**$*$. Este tipo de propriedades associada aos tipos básicos é de particular interesse para este trabalho, na medida em que as associações de ambientes em paralelo ou a aplicação do operador de replicação permitem reconhecer possíveis interferências na utilização de um identificador do tipo básico. Volta-se a este assunto mais à frente, na definição das regras para esse tipo de junção de ambientes.

A junção de dois ambientes em sequência une os ambientes encapsulando os tipos de cada um dos identificadores numa sequência de tipos. Suponham-se os ambientes Δ_1 e Δ_2 com um identificador x associado, respectivamente, aos tipos τ_1 e τ_2. O ambiente resultado da junção dos dois ambientes em sequência, $\Delta_1; \Delta_2$, irá conter a associação do identificador x para um novo tipo, $\tau_1; \tau_2$, correspondente ao encapsulamento dos dois tipos numa sequência.

A junção de dois ambientes numa escolha funciona de forma análoga à sequência, trocando o operador ; pelo operador $\&$.

$$\Delta_1 \,;\, \Delta_2 = \bigcup_{x \in D} x : (\tau_1 \,;\, \tau_2) \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array} \qquad \text{(Sequência)}$$

$$\Delta_1 \,\&\, \Delta_2 = \bigcup_{x \in D} x : (\tau_1 \,\&\, \tau_2) \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array} \qquad \text{(Escolha)}$$

Verificação de Interferências Indesejadas A verificação de interferências indesejadas está ligada às abstracções da linguagem para suportar concorrência. Intuitivamente, reconhece-se a existência de interferências a partir da identificação de uma escrita em paralelo com outras escritas ou leituras para o mesmo identificador.

As várias combinações entre tipos referência (**ref**(τ)), sincronizados (**sync**(τ)) ou simples constantes (τ) podem levar a que um tipo esteja ou não sujeito a uma interferência. Para auxiliar nessa verificação foram criadas duas funções auxiliares que reconhecem se um tipo tem referências e se é ou não um tipo sincronizado.

Um tipo tem referências se for uma referência directa para outro tipo ou, se for um tipo associado a um operador binário, tiver referências num dos lados do operador.

$$\mathsf{hasRefs}(\tau) = \begin{cases} \textbf{true}, & \text{if } \tau = \textbf{ref}(\tau') \\ \mathsf{hasRefs}(\tau_1) \vee \mathsf{hasRefs}(\tau_2), & \text{if } \tau = \tau_1 \; op \; \tau_2, \; op \in \{ \; ; \; , \; | \; , \; \& \; \} \end{cases} \qquad \text{(Referências)}$$

Um tipo está sincronizado se for uma sincronização directa de outro tipo ou, se for um tipo associado a um operador binário, ambos os lados do operador também forem sincronizados.

$$\mathsf{isSync}(\tau) = \begin{cases} \textbf{true}, & \text{if } \tau = \textbf{sync}(\tau') \\ \mathsf{isSync}(\tau_1) \wedge \mathsf{isSync}(\tau_2), & \text{if } \tau = \tau_1 \; op \; \tau_2, \; op \in \{ \; ; \; , \; | \; , \; \& \; \} \end{cases} \qquad \text{(Sincronização)}$$

As operações sobre ambientes que podem levar à existência de interferências indesejadas são as efectuadas pelos operadores de composição paralela e replicação.

O operador de replicação aplicado a um tipo τ modela uma utilização que pode ir desde a sua utilização zero, uma ou mais vezes em paralelo. Assim, se se tem $\tau!$, uma possibilidade de utilização será $\tau|\tau$. Essa utilização pode envolver a existência de interferências e por isso, antes de ser aplicado o operador ! a um ambiente, devem ser feitas algumas verificações.

Para o operador de replicação poder ser aplicado a um ambiente é preciso garantir que o tipo dos identificadores não define nenhuma escrita, ou definindo, essa escrita está protegida por um método sincronizado. Se essas garantias forem dadas então não existem interferências para nenhum dos identificadores desse ambiente.

$$\Delta! = \bigcup_{x \in D} x : \tau! \text{ , if } \neg\mathsf{hasRefs}(\tau) \; \vee \; \mathsf{isSync}(\tau) \qquad \begin{array}{c} D = Dom(\Delta) \\ \tau = \Delta(x) \end{array} \qquad \text{(Replicação)}$$

```
interface Pair {
  void setA(n:int);
  void setB(n:int);
  /*...*/

  usage{ /*...*/ (setA|setB) /*...*/ }
}


class PairImpl(a:int, b:int) implements Pair {
  void setA(n:int) {a := n} /*a:ref(int)*/
  void setB(n:int) {a := n} /*a:ref(int)*/
  /*...*/
}
```

Listagem 3.2: Reconhecimento de interferência através da junção de ambientes

A junção de ambientes pelo operador paralelo também efectua verificações com o mesmo propósito do operador de replicação. No entanto, a verificação para este tipo de junção é ligeiramente mais complexa. Como a junção é feita entre dois ambientes distintos, existem mais possibilidades a considerar na verificação de interferências indesejadas.

Se para um mesmo identificador x, um dos ambientes (Δ_1 ou Δ_2) não contém qualquer informação, então não é necessário efectuar nenhuma verificação. Significa que o identificador apenas foi usado numa das *threads* paralelas e, portanto, não sofre interferência da outra.

Já um mesmo identificador x para o qual existam informações guardadas nos dois ambientes (Δ_1 ou Δ_2) tem que garantir a inexistência de interferências entre ambos. Neste caso, um identificador x está livre de interferências na junção de ambientes se não houver escritas e leituras (ou apenas escritas) concorrentes em nenhum dos ambientes, ou havendo, ambas estão protegidas por métodos sincronizados.

$$\Delta_1 \mid \Delta_2 = \bigcup_{x \in D} x : \tau \qquad \begin{array}{c} D = Dom(\Delta_1) \cup Dom(\Delta_2) \\ \tau_1 = \Delta_1(x) \\ \tau_2 = \Delta_2(x) \end{array} \qquad \text{(Paralelo)}$$

$$\tau = (\tau_1 \mid \tau_2), \text{ if } \mathsf{checkParallel}(\tau_1, \tau_2) \ \vee \ (\tau_1 = \mathbf{void}) \ \vee \ (\tau_2 = \mathbf{void})$$

$$\mathsf{checkParallel}(\tau_1, \tau_2) = (\neg \mathsf{hasRefs}(\tau_1) \wedge \neg \mathsf{hasRefs}(\tau_2)) \ \vee \ (\mathsf{isSync}(\tau_1) \wedge \mathsf{isSync}(\tau_2))$$

Este tipo de verificações é válido para todas as variáveis de tipos básicos (γ). A afectação para objectos não é tratada neste trabalho, pois envolveria outro tipo de estudos relacionados com *aliasing* e *ownership* e que saem fora do âmbito definido.

No exemplo da listagem 3.2 pode verificar-se como a junção de ambientes ajuda a reconhecer erros na especificação de um programa. Neste caso é reconhecido um erro do programador que, por engano, ao contrário do que seria esperado, usou a variável de instância `a` no método `setB`.

Quando fosse feita a tipificação da classe `PairImpl` em relação a cada um dos seus métodos obtinham-se os seguintes ambientes:

$$\Delta_{setA} = \{a : \mathbf{ref}(\mathbf{int})\} \qquad e \qquad \Delta_{setB} = \{a : \mathbf{ref}(\mathbf{int})\}$$

que, unidos pelo respectivo protocolo de utilização tal como definido na interface `Pair` (**`usage`**`{(setA|setB)}`), tentaria efectuar a seguinte operação de junção (apresentada sem restrições):

$$\Delta_{setA} | \Delta_{setB} = \{a : \mathbf{ref}(\mathbf{int}) \mid \mathbf{ref}(\mathbf{int})\}$$

Como é reconhecida a interferência entre duas escritas em paralelo, o programa não é considerado correcto.

3.2.5 Regras de Tipificação

A partir das construções e regras auxiliares anteriores já é possível escrever a função de tipificação de um programa. Os vários casos da função estão descritos nas figuras 3.7, 3.8 e 3.9.

Descreve-se de seguida uma função de verificação genérica para um operador binário ϕ, arbitrário, para introduzir a notação usada.

$$\begin{aligned}
&\text{if } e = (e_1 \ \phi \ e_2) \\
&\quad \text{and:} \\
&\qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\
&\qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\
&\mathsf{check}(e, C) = (\tau, \Delta_1 \cup \Delta_2 \cup \Delta', C_2 \cup C')
\end{aligned}$$

Neste caso a expressão e é verificada em dois passos intermédios. Da avaliação de cada um dos passos, resultam os respectivos triplos que são depois combinados no resultado final, conforme o tipo de verificação que se deseje efectuar. O tipo resultado da verificação da expressão e é τ, o ambiente será uma junção de Δ_1 com Δ_2 e eventualmente outro ambiente desejado (Δ') e o conjunto de restrições é constituído por C_2 que propaga as restrições de C e C_1, unido com outro tipo de restrições próprias (C') referentes ao operador ϕ e que sejam necessárias para garantir a correcta tipificação da expressão e.

Omitem-se as explicações de algumas regras que seguem uma abordagem *standard* e são verificadas a partir de regras avaliadas como descrito acima.

Sequências de Expressões e Operador Paralelo As sequências de expressões propagam o conjunto de restrições desde o conjunto inicial, passando pela avaliação da expressão do lado esquerdo, terminando com a adição das restrições obtidas da verificação da expressão do lado direito. É feita a união dos ambientes em sequência e o tipo de retorno é o tipo da expressão do lado direito.

if $e = (e_1 \textbf{ and } e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\textbf{boolean}, (\Delta_1; \Delta_2), C_2 \cup \{\tau_1 <: \textbf{boolean}, \tau_2 <: \textbf{boolean}\})$
Usa-se a mesma regra para o operador **or**.

if $e = (e_1 + e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\textbf{int}, (\Delta_1; \Delta_2), C_2 \cup \{\tau_1 <: \textbf{int}, \tau_2 <: \textbf{int}\})$
Usa-se a mesma regra para os operadores $\{-, *, /\}$.

if $e = (e_1 < e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\textbf{boolean}, (\Delta_1; \Delta_2), C_2 \cup \{\tau_1 <: \textbf{int}, \tau_2 <: \textbf{int}\})$
Usa-se a mesma regra para os operadores $\{<=, >, >=\}$.

if $e = (e_1 == e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\textbf{boolean}, (\Delta_1; \Delta_2), C_2 \cup \{\})$
Usa-se a mesma regra para o operador $\{! =\}$.

if $e = (\textbf{not } e')$
  and:
    $(\tau, \Delta, C') = \mathsf{check}(e', C)$
$\mathsf{check}(e, C) = (\textbf{boolean}, \Delta, C' \cup \{\tau <: \textbf{boolean}\})$

if $e = (\textbf{out}(e'))$
  and:
    $(\tau, \Delta, C') = \mathsf{check}(e', C)$
$\mathsf{check}(e, C) = (\textbf{void}, \Delta, C')$

Figura 3.7: Regras de tipificação (1)

if $e = (e_1; e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\tau_2, (\Delta_1; \Delta_2), C_2)$

if $e = (e_1|e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\mathbf{void}, (\Delta_1|\Delta_2), C_2)$

if $e = (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, C)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
    $(\tau_3, \Delta_3, C_3) = \mathsf{check}(e_3, C_2)$
$\mathsf{check}(e, C) = (\tau_2, (\Delta_1; (\Delta_2 \& \Delta_3)), C_3 \cup \{\tau_1\ <: \mathbf{boolean}, \tau_2\ <:>\ \tau_3\})$

if $e = (\mathbf{while}\ e_1\ \mathbf{do}\ e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
$\mathsf{check}(e, C) = (\mathbf{void}, (\Delta_1; (\Delta_2; \Delta_1)*), C_2 \cup \{\tau_1\ <: \mathbf{boolean}\})$

if $e = (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$
  and:
    $(\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C)$
    $(\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1)$
    $\Delta_2' = \Delta_2 \backslash \{x\}$
$\mathsf{check}(e, C) = (\tau_2, (\Delta_1; \Delta_2'), C_2 \cup \tau_1 <: \Delta_2(x))$

if $e = x$
  and:
$\mathsf{check}(x) = (X, x : X, \emptyset)$ with X fresh

if $e = (x := e)$
  and:
    $(\tau, \Delta, C') = \mathsf{check}(e, C)$
    $x : \mathbf{ref}(X)$
    $\Delta' = \Delta; x : \mathbf{ref}(X)$ with X fresh
$\mathsf{check}(e, C) = (\mathbf{void}, \Delta', C' \cup \{\tau <: X\})$

Figura 3.8: Regras de tipificação (2)

if $e = (\textbf{new}\ A(e_1, ..., e_n))$
  and:
    $(\tau_i, \Delta_i, C_i) = \mathsf{check}(e_i, C_{i-1}) \quad \forall_{i \in 1...n}$
    $\tau = [\tau_1, ..., \tau_n] \Rightarrow X \quad$ with X fresh
    $\Delta = A : \tau \ |\Delta_1| \cdots |\Delta_n$
$\mathsf{check}(e, C_0) = (X, \Delta, C_n)$

if $e = (x.m(e_1, ..., e_n))$
  and:
    $(\tau_i, \Delta_i, C_i) = \mathsf{check}(e_i, C_{i-1}) \quad \forall_{i \in 1...n}$
    $\tau = (\tau_1, ..., \tau_n) \rightarrow X \quad$ with X fresh
    $\Delta = x : m : \tau \ |\Delta_1|...|\Delta_n|$
$\mathsf{check}(e, C_0) = (X, \Delta, C_n)$

if $e = (A(\overline{a : \delta})\ implements\ \{I\}\{\overline{M}\})$
  and:
    $(\tau_{r_i}, \Delta_i, C_i) = \mathsf{check}(M_i, C_{i-1}) \quad \forall_{i \in 1..n}$
    $(\tau_c, \Delta) = \mathsf{usageToType}(I, (\overline{\tau_{r_i}, \Delta_i}))$
    $C_f = C_n \cup \{\delta_j <: \Delta(a_j)\} \quad \forall_{j \in 1..p}$
    $\Delta \backslash_{\{\overline{a}\}} = \emptyset$
$\mathsf{check}(e, C_0) = ([\overline{\delta}] \Rightarrow \tau_c, \emptyset, \emptyset)$

Figura 3.9: Regras de tipificação (3)

$$\begin{array}{l}\text{if } e = (e_1; e_2) \\ \quad \text{and:} \\ \qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\ \qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\ \mathsf{check}(e, C) = (\tau_2, (\Delta_1; \Delta_2), C_2)\end{array} \quad \text{(Sequência)}$$

O operador paralelo segue regras de tipificação idênticas ao operador sequencial, mas o seu tipo resultado e a operação de junção de ambientes são naturalmente diferentes. Neste caso o tipo de retorno é **void** porque, não existindo obrigatoriedade dos dois lados do operador paralelo serem do mesmo tipo, não é possível prever o tipo de retorno. Quanto aos ambientes, a junção é feita pela operação de junção em paralelo ($\Delta_1|\Delta_2$).

$$\begin{array}{l}\text{if } e = (e_1|e_2) \\ \quad \text{and:} \\ \qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\ \qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\ \mathsf{check}(e, C) = (\mathbf{void}, (\Delta_1|\Delta_2), C_2)\end{array} \quad \text{(Paralelo)}$$

Expressão Condicional A verificação da expressão condicional apresenta uma variante em relação ao tradicional para incluir as operações de junção de ambientes. Os passos em tempo de execução da expressão **if** são simulados na junção dos ambientes: inicialmente é avaliada a expressão da condição (Δ_1) e de seguida (;) opta-se (&) pelo ramo correspondente ao valor da condição ($\Delta_2 \& \Delta_3$).

Define-se também que a expressão condicional deve devolver o mesmo tipo quer seja executado o ramo correspondente à condição verdadeira, quer seja executado o ramo correspondente à condição falsa: restrição $\tau_2 \ <:> \ \tau_3$ (τ_2 é subtipo de τ_3 e τ_3 é subtipo de τ_2).

$$\begin{array}{l}\text{if } e = (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, C) \\ \quad \text{and:} \\ \qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\ \qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\ \qquad (\tau_3, \Delta_3, C_3) = \mathsf{check}(e_3, C_2) \\ \mathsf{check}(e, C) = (\tau_2, (\Delta_1; (\Delta_2 \& \Delta_3)), C_3 \cup \{\tau_1 \ <: \mathbf{boolean}, \tau_2 \ <:> \ \tau_3\})\end{array}$$
(Expressão Condicional)

Ciclo À semelhança da expressão condicional, a expressão do ciclo também simula a execução do mesmo. Inicialmente é testada a condição (Δ_1). Se a condição for verdadeira, é verificada a expressão do ciclo (Δ_2) e testada novamente a condição do ciclo (Δ_1). O processo é repetido (operador $*$) enquanto a condição for verdadeira, obtendo-se o ambiente final: $\Delta_1; (\Delta_2; \Delta_1)*$.

Como não se sabe se o ciclo vai ser executado alguma vez, o tipo resultado é **void**.

$$\begin{array}{l}\text{if } e = (\textbf{while } e_1 \textbf{ do } e_2) \\ \quad \text{and:} \\ \qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\ \qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\ \mathsf{check}(e, C) = (\textbf{void}, (\Delta_1; (\Delta_2; \Delta_1)*), C_2 \cup \{\tau_1 <: \textbf{boolean}\})\end{array} \quad \text{(Ciclo)}$$

Declaração Local As declarações locais associam um identificador (x) a uma constante de um determinado tipo (τ_1). O identificador x pode ser usado ao longo da expressão e_2, sendo a sua utilização guardada no ambiente Δ_2. Depois de verificadas as duas expressões importa garantir que o tipo da constante está de acordo com a utilização que foi feita do respectivo identificador. Para isso, adiciona-se essa restrição ($\tau_1 <: \Delta_2(x)$) ao conjunto de restrições propagado C_2.

O ambiente resultado corresponde à junção sequencial dos ambientes Δ_1 e Δ_2, retirando (filtrando) a informação referente ao identificador x, cuja verificação já foi adicionada ao conjunto de restrições.

$$\begin{array}{l}\text{if } e = (\textbf{let } x = e_1 \textbf{ in } e_2) \\ \quad \text{and:} \\ \qquad (\tau_1, \Delta_1, C_1) = \mathsf{check}(e_1, C) \\ \qquad (\tau_2, \Delta_2, C_2) = \mathsf{check}(e_2, C_1) \\ \qquad \Delta_2' = \Delta_2 \backslash \{x\} \\ \mathsf{check}(e, C) = (\tau_2, (\Delta_1; \Delta_2'), C_2 \cup \tau_1 <: \Delta_2(x))\end{array} \quad \text{(Declaração Local)}$$

Uma das limitações da linguagem está presente nas declarações locais: não é possível fazer *aliasing* de objectos. A listagem 3.3 ilustra um caso de erro de tipificação devido ao aliasing entre os identificadores `x` e `y`. Para que fosse possível um identificador referir um objecto previamente criado, teria que se encontrar uma forma de combinar as utilizações para os dois identificadores distintos. A forma de combinar esses identificadores levaria a um estudo complexo, nomeadamente no âmbito da *ownership* de objectos, fora do âmbito deste trabalho.

Identificador Quando é encontrado um identificador, esse identificador é associado a variável de tipo, X, "fresca" pois não se conhece o tipo do identificador.

Como não é possível fazer qualquer tipo de considerações sobre o tipo do identificador, não são adicionadas quaisquer tipo de restrições.

$$\begin{array}{l}\text{if } e = x \\ \quad \text{and:} \\ \mathsf{check}(x) = (X, x : X, \emptyset) \quad \text{with } X \text{ fresh}\end{array} \quad \text{(Identificador)}$$

Afectação Para verificar a correcta tipificação de uma afectação ($x := e$) é associado a x o tipo referência (**ref**) para uma nova variável de tipo X, fresca. Por sua vez, é adicionada uma

```
interface T {
  void m1();
  void m2();
  void m3();

  usage{ m1;m2;m3 }
}

class C implements T {
  void m1() { out(1) }
  void m2() { out(2) }
  void m3() { out(3) }
}

main {
  let x = new C() in /*x:X;m1;m3*/
  let y = x in { /*y:m2, x:X*/
    x.m1();y.m2();x.m3() /*x:m1;m3, y:m2*/
  }
}
```

Listagem 3.3: Gestão de aliasing

restrição ao tipo X, indicando que este deve ser supertipo do tipo τ (tipo da expressão e), ou seja, garante-se que não se está a passar para o identificador x um tipo com menor capacidade do que o tipo esperado desse identificador.

A construção **ref** indica que o identificador x será uma referência para um tipo e, ao mesmo tempo, que x foi objecto de uma afectação (escrita). A marcação de x como sendo uma variável que sofreu uma afectação é de particular importância para a verificação da existência (ou não) de interferências indesejadas.

$$\begin{array}{l} \text{if } e = (x := e) \\ \quad \text{and:} \\ \qquad (\tau, \Delta, C') = \mathsf{check}(e, C) \\ \qquad x : \mathbf{ref}(X) \\ \qquad \Delta' = \Delta; x : \mathbf{ref}(X) \quad \text{with } X \text{ fresh} \\ \mathsf{check}(e, C) = (\mathbf{void}, \Delta', C' \cup \{\tau <: X\}) \end{array} \qquad \text{(Afectação)}$$

Instanciação A notação seguida para as instâncias associa ($\Rightarrow$) os tipos dos argumentos ($[\tau_1, ..., \tau_n]$) à utilização feita dessa instância (X).

A criação de novas instâncias limita-se a fazer a verificação de cada um dos argumentos do novo objecto, criando uma nova associação entre os tipos de argumento obtidos e a variável de tipo, X, fresca. Esse tipo X corresponde ao tipo da instância. Por sua vez, o tipo da instância corresponde à utilização (chamada de métodos) que foi feita a essa instância. É a partir desta utilização que são comparados os tipos das instâncias pelo algoritmo de simulação.

O ambiente resultado corresponde à junção em paralelo dos ambientes resultantes da verificação dos argumentos com a associação do nome da instância (A) ao seu tipo correspondente ($[\tau_1, ..., \tau_n] \Rightarrow X$). A junção dos ambientes é feita em paralelo e não em sequência pois não é possível desassociar a verificação dos argumentos da criação da nova instância.

$$\begin{array}{l}\text{if } e = (\textbf{new } A(e_1, ..., e_n)) \\ \quad \text{and:} \\ \qquad (\tau_i, \Delta_i, C_i) = \mathsf{check}(e_i, C_{i-1}) \quad \forall_{i \in 1...n} \\ \qquad \tau = [\tau_1, ..., \tau_n] \Rightarrow X \quad \text{with } X \text{ fresh} \\ \qquad \Delta = A : \tau \; |\Delta_1| \cdots |\Delta_n \\ \mathsf{check}(e, C_0) = (X, \Delta, C_n)\end{array} \qquad \text{(Instanciação)}$$

Chamada de Método A verificação da chamada de método segue uma abordagem semelhante à instanciação. É criada uma associação entre os tipos dos argumentos e um tipo indeterminado X, fresco. O tipo X corresponde ao tipo de retorno do método.

A junção dos ambientes é também efectuada em paralelo, associando do identificador x (da instância) a uma chamada do método m, do tipo τ. O tipo τ corresponde à associação ($\rightarrow$) entre os argumentos ($\tau_1, ..., \tau_n$) do método e o seu tipo de retorno (X).

O resultado da verificação da chamada de método é, portanto, o triplo constituído pelo tipo indeterminado X, a junção de ambientes descrita acima e o conjunto de restrições propagadas resultante da verificação de cada um dos argumentos do método.

$$\begin{array}{l}\text{if } e = (x.m(e_1, ..., e_n)) \\ \quad \text{and:} \\ \qquad (\tau_i, \Delta_i, C_i) = \mathsf{check}(e_i, C_{i-1}) \quad \forall_{i \in 1...n} \\ \qquad \tau = (\tau_1, ..., \tau_n) \rightarrow X \quad \text{with } X \text{ fresh} \\ \qquad \Delta = x : m : \tau \; |\Delta_1| ... |\Delta_n| \\ \mathsf{check}(e, C_0) = (X, \Delta, C_n)\end{array} \qquad \text{(Chamada de Método)}$$

Método A verificação de um método adiciona a restrição de que o tipo do corpo do método (τ) seja subtipo do tipo definido como retorno do método (τ_r) e uma restrição para cada tipo de cada um dos parâmetros definidos (τ_i), indicando que devem ser subtipo da utilização que lhes foi feita ($\Delta(x_i)$). Este tipo de restrições seguem as habituais regras de covariância e contravariância [Cas95].

Ao ambiente resultado da verificação do corpo do método são retiradas as ocorrências referentes aos parâmetros do mesmo (filtradas depois de adicionadas as respectivas restrições – $\Delta \backslash \{\overline{x}\}$). O ambiente resultado poderá apenas conter associações para *fields* da classe, verificação que será efectuada ao nível da mesma.

$$
\begin{array}{l}
\text{if } e = (\tau_r\ m(\overline{x:\tau})\{e'\}) \\
\quad \text{and:} \\
\qquad (\tau, \Delta, C') = \mathsf{check}(e', C) \\
\qquad C_f = C' \cup \{\tau <: \tau_r, \tau_i <: \Delta(x_i)\} \quad \forall_{i \in 1..n} \\
\mathsf{check}(e, C) = (m : (\tau_1, ..., \tau_n) \to \tau_r, \Delta \backslash \{\overline{x}\}, C_f)
\end{array}
\qquad \text{(Método)}
$$

Classe A verificação das classes está associada às verificações de cada um dos seus métodos. É utilizada uma construção auxiliar que transforma o protocolo de utilização definido na interface (I), à custa dos nomes dos métodos, em tipos que possam ser verificados: os tipos dos métodos. Depois de verificados todos os métodos é utilizada essa construção auxiliar (figura 3.5) para agregar os tipos dos métodos segundo o protocolo de utilização definido, sendo obtido o tipo da classe, τ_c, e o respectivo ambiente, Δ.

A partir do tipo da classe e do seu ambiente, são verificados os *fields* da classe, cuja utilização ($\Delta(a_j)$) deve estar de acordo com o tipo com que foram definidos (δ_j).

Se todas as restrições (C_f) forem coerentes e não sobrar qualquer identificador livre depois de filtrados os identificadores dos *fields* da classe, então a classe considera-se bem tipificada.

$$
\begin{array}{l}
\text{if } e = (A(\overline{a:\delta})\ implements\ \{I\}\{\overline{M}\}) \\
\quad \text{and:} \\
\qquad (\tau_{r_i}, \Delta_i, C_i) = \mathsf{check}(M_i, C_{i-1}) \quad \forall_{i \in 1..n} \\
\qquad (\tau_c, \Delta) = \mathsf{usageToType}(I, (\overline{\tau_{r_i}, \Delta_i})) \\
\qquad C_f = C_n \cup \{\delta_j <: \Delta(a_j)\} \quad \forall_{j \in 1..p} \\
\qquad \Delta \backslash_{\{\overline{a}\}} = \emptyset \\
\mathsf{check}(e, C_0) = ([\overline{\delta}] \Rightarrow \tau_c, \emptyset, \emptyset)
\end{array}
\qquad \text{(Classe)}
$$

Verificação de Restrições O conjunto de restrições C vai sendo propagado e verificado à medida que se avança na verificação e um programa. Se a união de um conjunto de restrições C_1 com um conjunto de restrições C_2 ($C_1 \cup C_2$) se verificar incompatível então é dado um erro de verificação e o programa é considerado incorrecto.

Se não se verificar qualquer incompatibilidade nos conjuntos de restrições e no final da verificação não houver identificadores livres num programa então o programa é considerado correcto.

3.3 Implementação

A linguagem FWCJ é uma linguagem interpretada, que funciona sobre a máquina virtual do Java. O fluxo de um programa decorre tal como ilustrado na figura 3.10.

Cada programa é construído a partir de um ficheiro (único) com o código do programa na linguagem FWCJ que é analisado pelo parser que transporta o código para uma árvore de

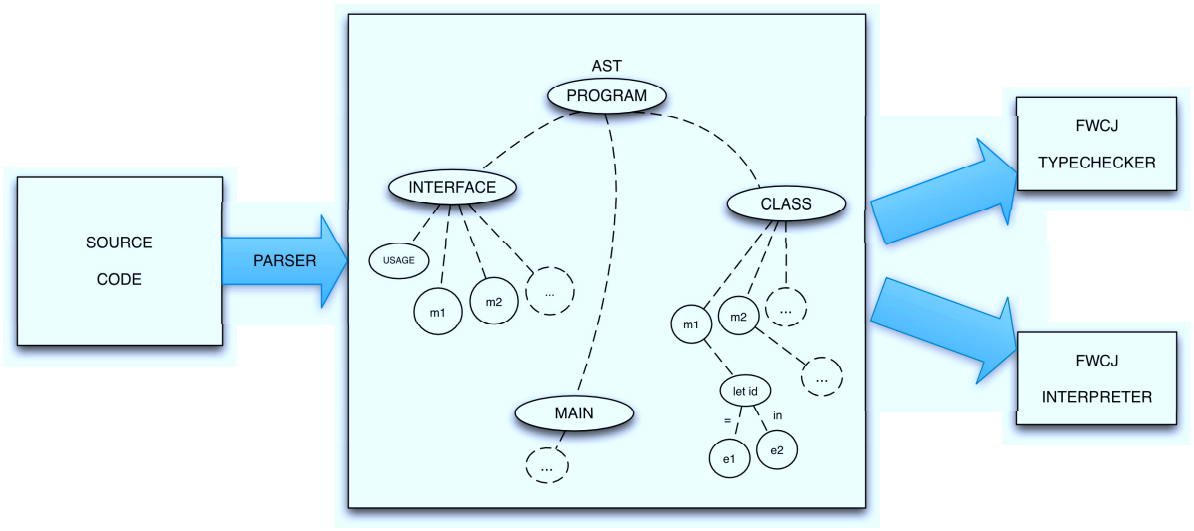

Figura 3.10: Fluxo geral de um programa

sintaxe abstracta (AST). O *parser* foi definido em Java CUP [1] com um gerador de código Java, o JFlex [2], a partir do qual se constrói a AST de um programa implementado na linguagem FWCJ.

O algoritmo de verificação e o sistema de *runtime* são conseguidos através da análise da AST do FWCJ. Essa análise é feita a partir de *visitors* genéricos que facilmente podem ser alterados para verificar outras propriedades.

Foi também criado um pequeno plugin para a plataforma Eclipse (figura 3.11) que ajuda na diferenciação e identificação dos ficheiros com a extensão `fwcj`, faz *syntax highlighting* das palavras e tipos reservados da linguagem e auxilia na identificação de erros de tipificação.

3.3.1 Opções Tomadas e Principais Desafios

A decisão de gerar código para a linguagem Java deveu-se ao maior conhecimento da mesma, adquirido ao longo do percurso académico e o *parser* foi implementado através da ferramenta CUP que apresentava uma boa possibilidade de integração com o Java.

A definição do algoritmo de simulação foi um dos principais desafios deste trabalho. Ainda que as regras de transição e do algoritmo de simulação tenham sido definidas de forma simples e directa, a sua implementação não foi trivial. A comparação dos conjuntos de frentes, a verificação das várias possibilidades de transição e das várias relações de *subtyping* possíveis após cada transição foram das principais dificuldades no decorrer da implementação. Apesar da dificuldade na implementação do mesmo e das várias abordagens e alterações que foram sendo feitas até ao resultado final, pensa-se ter chegado a uma solução estável e de fácil compreensão para quem ler este documento. Houve o cuidado de comentar a maior parte do código para a linguagem se torne extensível, eventualmente, para serem verificadas outras propriedades que

[1] http://www.cs.princeton.edu/~appel/modern/java/CUP/
[2] http://jflex.de/

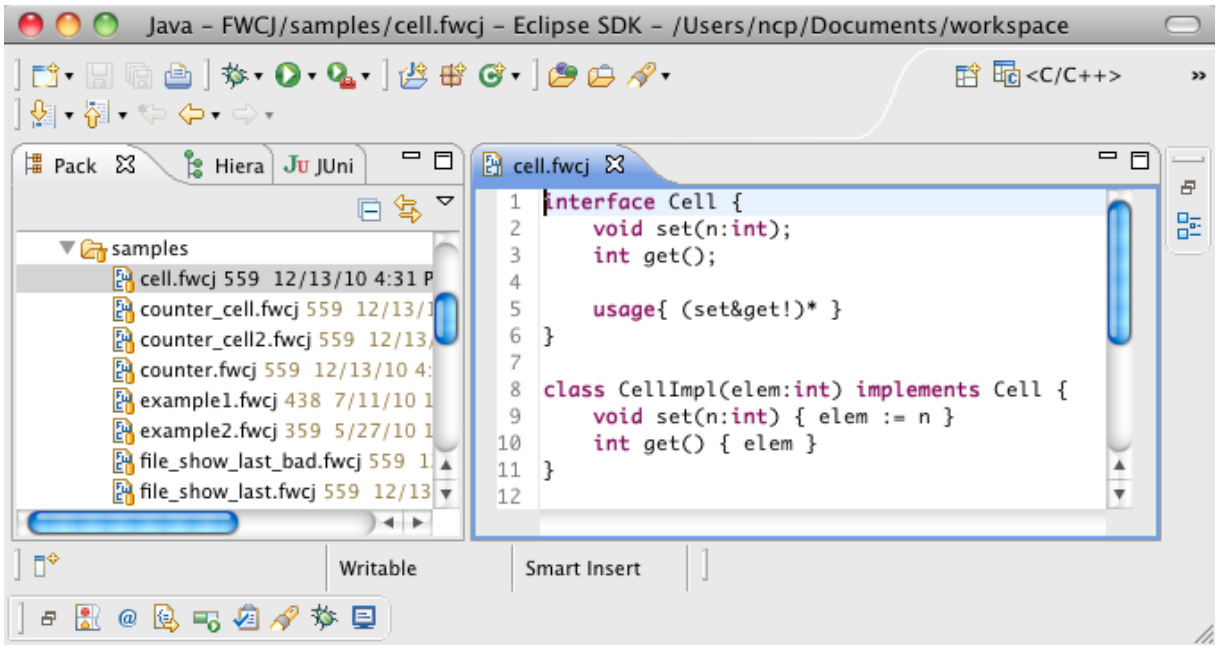

Figura 3.11: *Plugin* para a plataforma eclipse

não foram tratadas neste trabalho.

3.3.2 Verificação de Correcção

A verificação da correcção do algoritmo de tipificação apresentado foi feita a partir da implementação da linguagem FWCJ. Para auxiliar a verificação, foi criada uma bateria de testes que pode ser consultada no anexo A. Nesse conjunto de teste podem encontrar-se vários exemplos de programas bem tipificados e programas mal tipificados.

4

Considerações Finais

O objectivo desta dissertação consistiu na definição de um algoritmo de tipificação para uma linguagem com suporte para abstracções próprias que tratam o controlo de concorrência. A introdução desse tipo de abstracções retiram ao programador a necessidade de, ele próprio, ter que especificar os mecanismos para efectuar controlo de concorrência. Assim, a definição de um programa é feita de uma forma mais simples, deixando à linguagem a responsabilidade de gerir o controlo de concorrência.

A linguagem desenvolvida é capaz de lidar com programas concorrentes, verificando estaticamente a existência de *race conditions* em tempo de execução (programa mal tipificado). Prevêem-se, portanto, possíveis comportamentos inesperados de um programa concorrente em que várias *threads* interagem entre si.

4.1 Contribuições

A linguagem desenvolvida define um algoritmo de tipificação baseado em tipos espaciais/-comportamentais, com operadores de tipo de composição sequencial, paralela, escolha, repetição e replicação. O algoritmo é apresentado na forma de um algoritmo de simulação, com regras bastante intuitivas para lidar com esses mesmos tipos. São ainda combinadas técnicas de inferência de tipos com relações de *subtyping* de forma a tornar a especificação de um programa menos pesada para os programadores, diminuindo o número de anotações de tipos necessárias.

A validação do algoritmo foi feita através da definição de um protótipo da linguagem (interpretada) e respectivo *typechecker* ao qual foram submetidos vários programas de exemplo,

para testar o maior número de casos possíveis.

Assim, na sequência deste trabalho, é apresentada a linguagem FWCJ – Featherweight Concurrent Java e a definição do seu algoritmo de tipificação. É também disponibilizado o referido protótipo da linguagem e respectiva implementação como forma de possibilitar uma aplicação real do estudo desta dissertação.

Principais Dificuldades Apesar dos casos do algoritmo de tipificação serem simples, não foi fácil defini-los até chegar ao resultado final, tal como apresentado no capítulo 3. Antes de se chegar à solução final, foram testadas outras definições do algoritmo que, apesar de atingirem o propósito deste trabalho, se verificavam demasiado complexas e nem sempre bem definidas. Em alguns casos particulares, como os tipos associados ao operador de repetição e, principalmente, aos tipos associados ao operador de replicação, foi necessário verificar um grande número de propriedades. Como esses operadores introduzem uma noção de "infinitas" possibilidades de execução, o algoritmo de simulação para transições entre dois tipos sofreu várias iterações até se conseguir encontrar uma solução que lidasse com todos os casos.

4.2 Trabalho Futuro

A prova formal do algoritmo é um dos aspectos que poderá ser abordado no futuro para garantir a correcção deste trabalho.

No que diz respeito às limitações da linguagem, desde logo, o tratamento de situações de *aliasing* e *ownership* enriqueceriam ainda mais as capacidades da mesma, nomeadamente se fosse possível realizar afectações a tipos objecto. As afectações a tipos objecto, quando estes representam tipos comportamentais, não é uma operação trivial. A possibilidade de haver mais do que uma referência para o mesmo objecto levanta questões no âmbito da correcta utilização do mesmo, tal como definido no seu tipo. Se não for feita uma gestão cuidada de todas as referências do objecto, o protocolo definido no seu tipo pode ser quebrado.

A linguagem definida é uma linguagem interpretada e funciona sobre a máquina virtual do Java. O desenvolvimento de um compilador próprio, com suporte para as abstracções de concorrência, seria outro desafio interessante para a continuação deste trabalho.

Bibliografia

[Agh85] G Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, Jan 1985.

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, e Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pág. 1015–1022, New York, NY, USA, 2009. ACM.

[AVWW93] J. Armstrong, R. Virding, C. Wikstrom, e M. Williams. Concurrent programming in ERLANG. 1993.

[BBA08] N.E. Beckman, K. Bierhoff, e J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pág. 227–244. ACM New York, NY, USA, 2008.

[BLR02] C. Boyapati, R. Lee, e M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pág. 211–230. ACM New York, NY, USA, 2002.

[BLS05] Mike Barnett, Leino, e Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, pág. 49–69. Springer, Berlin / Heidelberg, Janeiro 2005.

[BMS05] G. Bierman, E. Meijer, e W. Schulte. The essence of data access in Cω. *ECOOP 2005-Object-Oriented Programming*, pág. 287–311, 2005.

[Bor10] R Bornat. Separation logic and concurrency. *Formal Methods: State of the Art and New Directions*, pág. 217–248, 2010.

[Cai08] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoretical Computer Science*, 402(2-3):120–141, 08 2008.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76:138–164, February 1988.

[Cas95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17:431–447, May 1995.

[CC03] L. Caires e L. Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194–235, 2003.

[CD02] David Clarke e Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pág. 292–310, November 2002.

[CDNS07] Nicholas Cameron, Sophia Drossopoulou, James Noble, e Matthew Smith. Multiple Ownership. In *OOPSLA 07*, September 2007.

[CM04] K.L. Clark e F.G. McCabe. Go! – A multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 41(2):171–206, 2004.

[CS10] Luís Caires e João Costa Seco. Dynamic control of interference. Relatório Técnico DIFCTUNL-5-2010, CITI / DI-FCT-UNL, December 2010.

[Dij65] E Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, Jan 1965.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Agosto 1975.

[DM82] Luis Damas e Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pág. 207–212, New York, NY, USA, 1982. ACM.

[FA99] C. Flanagan e M. Abadi. Types for safe locking. *Programming Languages and Systems*, pág. 640–640, 1999.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, e Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pág. 234–245, New York, NY, USA, 2002. ACM.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, e Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[Han76] Per Brinch Hansen. The programming language concurrent pascal. In *Language Hierarchies and Interfaces, International Summer School*, pág. 82–110, London, UK, 1976. Springer-Verlag.

[Han93] Per Brinch Hansen. Monitors and concurrent pascal: a personal history. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pág. 1–35, New York, NY, USA, 1993. ACM.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, e Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18:109–138, March 1996.

[HMPJH05] T. Harris, S. Marlow, S. Peyton-Jones, e M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pág. 48–60. ACM New York, NY, USA, 2005.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[IK02] G Stewart Itzstein e David Kearney. Applications of join java. *Aust. Comput. Sci. Commun.*, 24:37–46, January 2002.

[IPW99] Atsushi Igarashi, Benjamin C. Pierce, e Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pág. 132–146, 1999.

[LBR98] G.T. Leavens, A.L. Baker, e C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, 1998.

[Lei83] D. Leivant. Polymorphic type inference. *Proceedings of the 10th ACM SIGACT-SIGPLAN*, Jan 1983.

[LW94] Barbara H. Liskov e Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841, November 1994.

[MAC10] F. Militão, J. Aldrich, e L. Caires. Aliasing control with view-based typestate. *Proceedings of INFORUM 2009-Simpósio de Informática*, 2010.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil08] Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Tese de Mestrado, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2008.

[Mit91] J.C. Mitchell. Type inference with simple subtypes. *Journal of functional programming*, 1(03):245–285, 1991.

[MS96] M.M. Michael e M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pág. 275. ACM, 1996.

[Nel91] G. Nelson. *Systems programming with Modula-3*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1991.

[OSV08] M. Odersky, L. Spoon, e B. Venners. *Programming in Scala*. Artima Inc, 2008.

[Pir09] Margarida Piriquita. Type system for the componentj programming language. Tese de Mestrado, Universidade Nova de Lisboa – Faculdade de Ciências e Tecnologia, 2009.

[Pyl85] IC Pyle. The Ada programming language. *PRENTICE-HALL INTERNATIONAL(UK) LTD., LONDON(UK), 1985, 341*, 1985.

[Rep93] John Reppy. Concurrent ml: Design, application and semantics. In Peter Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pág. 165–198. Springer Berlin / Heidelberg, 1993.

[Rey78] John C. Reynolds. Syntactic control of interference. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pág. 39–46, New York, NY, USA, 1978. ACM.

[RLNS00] K. Rustan, M. Leino, Greg Nelson, e James B. Saxe. Esc/java user?s manual. Relatório técnico, Compaq Computer Corporation - Systems Research Center, 2000.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[San10] Tiago Santos. Linguagem de Especificação Leve Hoare-Separação para Java. Tese de Mestrado, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Julho 2010.

[SD03] Matthew Smith e Sophia Drossopoulou. Cheaper reasoning with ownership types. In *IWACO 2003 - Workshop affilicated to ECOOP 2003*, June 2003.

[Sta88] R. Stansifer. Type inference with subtypes. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pág. 88–97, New York, NY, USA, 1988. ACM.

[SY86] RE Strom e S. Yemini. Typestate- A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

[VM06] V.T. Vasconcelos e F. Martins. A multithreaded typed assembly language. *Proceedings of TV*, 6:133–141, 2006.

Exemplos de Teste para a Linguagem FWCJ

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

main {
  let x = new CellImpl(1) in { /*x:get;set;get*/
    out(x.get());
    x.set(5);
    out(x.get())
  }
}
```

Listagem A.1: Célula de memória

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

main {
  let x = new CellImpl(1) in {  /*x:(get|get);set*/
    (out(x.get())|out(x.get()));
    x.set(5)
  }
}
```

Listagem A.2: Célula de memória usada de forma incorrecta

```
interface Counter {
  void inc();
  int get();

  usage{ (inc&get!)* }
}

class CounterImpl(n:int) implements Counter {
  void inc() { n := n+1 } /*n:ref(int)*/
  int get() { n } /*n:int*/
}

main {
  let c = new CounterImpl(0) in { /*c:get;inc;inc;get*/
    out(c.get());
    c.inc();
    c.inc();
    out(c.get())
  }
}
```

Listagem A.3: Contador

```
interface Counter {
  void inc();
  int get();

  usage{ (inc&get!)* }
}

class CounterImpl(n:int) implements Counter {
  void inc() { n := n+1 } /*n:ref(int)*/
  int get() { n } /*n:int*/
}

interface Cell {
  void inc();
  int get();

  usage{ (inc&get!)* }
}

class CellImpl(elem:Counter) implements Cell {
  void inc() { elem.inc() } /*elem:inc*/
  int get() { elem.get() } /*elem:get*/
}


main {
  let c = new CounterImpl(0) in {
    let x = new CellImpl(c) in { /*x:get;inc;inc;get*/
      out(x.get());
      x.inc();
      x.inc();
      out(x.get())
    }
  }
}
```

Listagem A.4: Contador com base numa célula de memória

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

interface Counter {
  void inc();
  int get();

  usage{ (inc&get!)* }
}

class CounterImpl(x:Cell) implements Counter {
  void inc() { /*x:set*/
    let old = x.get() in {
      x.set(old+1)
    }
  }
  int get() { x.get() } /*x:get*/
}


main {
  let c = new CellImpl(3) in {
    let x = new CounterImpl(c) in { /*x:get;inc;inc;get*/
      out(x.get());
      x.inc();
      x.inc();
      out(x.get())
    }
  }
}
```

Listagem A.5: Outro contador com base numa célula de memória

```
interface Pair {
  void setA(n:int);
  void setB(n:int);
  int getA();
  int getB();

  usage{ (setA&setB&getA&getB)* }
}


class PairImpl(a:int, b:int) implements Pair {
  void setA(n:int) {a := n} /*a:ref(int)*/
  void setB(n:int) {b := n} /*b:ref(int)*/
  int getA() { a } /*a:int*/
  int getB() { b } /*b:int*/
}

main {
  let x = new PairImpl(1,2) in { /*x:getA;getB;setA;getA*/
    out(x.getA());
    out(x.getB());
    x.setA(5);
    out(x.getA())
  }
}
```

Listagem A.6: Par de inteiros (1)

```
interface Pair {
  void setA(n:int);
  void setB(n:int);
  int getA();
  int getB();

  usage{ (setA|setB&getA&getB)* }
}


class PairImpl(a:int, b:int) implements Pair {
  void setA(n:int) {a := n} /*a:ref(int)*/
  void setB(n:int) {b := n} /*b:ref(int)*/
  int getA() { a } /*a:int*/
  int getB() { b } /*b:int*/
}

main {
  out("Nao ha interferencia para a mesma variavel")
}
```

Listagem A.7: Par de inteiros (2)

```
interface Pair {
  void setA(n:int);
  void setB(n:int);
  int getA();
  int getB();

  usage{ (setA|setB&getA&getB)* }
}


class PairImpl(a:int, b:int) implements Pair {
  void setA(n:int) {a := n} /*a:ref(int)*/
  void setB(n:int) {a := n} /*a:ref(int)*/
  int getA() { a } /*a:int*/
  int getB() { b } /*b:int*/
}

main {
  let x = new PairImpl(1,2) in { /*x:getA;getB;setA;getA*/
    out(x.getA());
    out(x.getB());
    x.setA(5);
    out(x.getA())
  }
}
```

Listagem A.8: Erro na definição da classe de um par de inteiros

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

interface Pair {
  void setA(n:int);
  void setB(n:int);
  int getA();
  int getB();

  usage{ (setA&getA!)*|(setB&getB!)* }
}


class PairImpl(a:Cell, b:Cell) implements Pair {
  void setA(n:int) { a.set(n) } /*a:set*/
  void setB(n:int) { b.set(n) } /*b:set*/
  int getA() { a.get() } /*a:get*/
  int getB() { b.get() } /*b:get*/
}



main {
  let x = new CellImpl(1) in {
    let y = new CellImpl(2) in {
      let z = new PairImpl(x,y) in { /*z:get;setB;getB*/
        out(z.getA());
        z.setB(5);
        out(z.getB())
      }
    }
  }
}
```

Listagem A.9: Par com base numa célula de memória

```
interface File {
  void open();
  void read();
  void write(s:string);
  void close();
  usage{ open;(read! & write)*;close }
}

class FileImpl(file:string) implements File {
  void open() { out("abriu") }
  void write(s:string) { file := s } /*file:ref(string)*/
  void read() { out(file) } /*file:int*/
  void close() { out("fechou") }
}

main {
  let x = new FileImpl("novo ficheiro") in { /*x:open;read;write;read;close*/
    x.open();
    x.read();
    x.write("nova escrita");
    x.read();
    x.close()
  }
}
```

Listagem A.10: Ficheiro

```
interface File {
  void open();
  void read();
  void write(s:string);
  void close();
  usage{ open;(read & write)*;read;close }
}

class FileImpl(file:string) implements File {
  void open() { out("abriu") }
  void write(s:string) { file := s }
  void read() { out(file) }
  void close() { out("fechou") }
}

main {
  let x = new FileImpl("novo ficheiro") in {
    x.open();
    x.read();
    x.write("nova escrita");
    /*x.read(); falha porque nao fez leitura final*/
    x.close()
  }
}
```

Listagem A.11: Utilização incorrecta de um ficheiro

```
interface File {
  void open();
  void read();
  void write(s:string);
  void close();
  usage{ open;(read & write)*;read;close }
}

class FileImpl(file:string) implements File {
  void open() { out("abriu") }
  void write(s:string) { file := s }
  void read() { out(file) }
  void close() { out("fechou") }
}

main {
  let x = new FileImpl("novo ficheiro") in {
    x.open();
    x.read();
    x.write("nova escrita");
    x.read(); /*faz leitura final e respeita o protocolo*/
    x.close()
  }
}
```

Listagem A.12: Utilização correcta de um ficheiro

```
interface Cell {
  void set1(n:int);
  void set2(n:int);
  int get();

  usage{ ((set1|set2)&get!)* }
}

class CellImpl(elem:int) implements Cell {
  sync void set1(n:int) { elem := n } /*elem:ref(int)*/
  sync void set2(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

main {
  let x = new CellImpl(1) in { /*x:get;(set1|set2);get*/
    out(x.get());
    (x.set1(5)|x.set2(2));
    out(x.get())
  }
}
```

Listagem A.13: Métodos sincronizados

```
interface Cell {
  void set(n:int);
  int get();

  usage{ (set&get!)* }
}

class CellImpl(elem:int) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
}

main {
  let c = new CellImpl(0) in { /*c:set*;get*/
    let x = 0 in {
      while (x < 10) do {
        x := x + 1;
        c.set(x)
      }
    };
    out(c.get())
  }
}
```

Listagem A.14: Ciclo *while* – aplicação do protocolo de repetição

```
interface T {
  void m1();
  void m2();
  void m3();

  usage{ m1;m2;m3 }
}

class C() implements T {
  void m1() { out(1) }
  void m2() { out(2) }
  void m3() { out(3) }
}

main {
  let x = new C() in /*x:X;m1;m3*/
  let y = x in { /*y:m2(), x:X*/
    x.m1();y.m2();x.m3() /*x:m1;m3, y:m2*/
  }
  /*aliasing nao e tratado*/
}
```

Listagem A.15: *Aliasing* (não é tratado – erro de tipificação)

```
interface IntList {
  int get(pos:int);
  IntList add (n:int);
  IntList remove(pos:int);

  usage{ (get!&add&remove)* }
}

class IntListImpl(head:int, tail:IntList) implements IntList {
  int get(pos:int){ /*head:int, tail:get*/
    if (pos == 0) then {
      head
    }
    else{
      tail.get(pos-1)
    }
  }

  IntList add (n:int){ /*head:int&int, tail:add*/
    if (tail == null) then {
      let newTail = new IntListImpl(n,null) in new IntListImpl(head,newTail)
    }
    else{
      let newTail = tail.add(n) in new IntListImpl(head,newTail)
    }
  }

  IntList remove (pos:int){ /*tail:IntList&remove, head:int*/
    if (pos == 0) then {
      tail
    }
    else{
      let newList = new IntListImpl(head,tail.remove(pos-1)) in newList
    }
  }
}


main {
  let list0 = new IntListImpl(0,null) in /*list0:add*/
  let list1 = list0.add(1) in /*list1:add*/
  let list2 = list1.add(2) in /*list2:add*/
  let list3 = list2.add(3) in /*list3:add*/
  let list4 = list3.add(4) in { /*list4:get;get;get;get;remove*/
    out(list4.get(1));
    out(list4.get(2));
    out(list4.get(4));
    out(list4.get(3));
    let newList = list4.remove(2) in { /*newList:get;get*/
      out(newList.get(2));
      out(newList.get(1))
    }
  }
}
```

Listagem A.16: Lista de inteiros

```
interface IntList {
  int get(pos:int);
  IntList add (n:int);
  IntList remove(pos:int);

  usage{ (get!&add&remove)* }
}

class IntListImpl(head:int, tail:IntList) implements IntList {
  int get(pos:int){ /*head:int, tail:get*/
    if (pos == 0) then {
      head
    }
    else{
      tail.get(pos-1)
    }
  }

  IntList add (n:int){ /*head:int&int, tail:add*/
    if (tail == null) then {
      let newTail = new IntListImpl(n,null) in new IntListImpl(head,newTail)
    }
    else{
      let newTail = tail.add(n) in new IntListImpl(head,newTail)
    }
  }

  IntList remove (pos:int){ /*tail:IntList&remove, head:int*/
    if (pos == 0) then {
      tail
    }
    else{
      let newList = new IntListImpl(head,tail.remove(pos-1)) in newList
    }
  }
}


main {
  let list0 = new IntListImpl(0,null) in
  let list1 = list0.add(1) in
  let list2 = list1.add(2) in
  let list3 = list2.add(3) in
  let list4 = list3.add(4) in { /*list4:add|add|get|get viola o protocolo*/
    list2.add(1)|list2.get(2)|list4.get(2)|list4.add(3)
  }
}
```

Listagem A.17: Lista de inteiros mal especificada

```
interface Cell {
  void set(n:int);
  int get();
  Cell getNext();
  Cell createIntBuffer(aux:Cell, size:int);


  usage{ createIntBuffer* & (set&getNext&get!)* }
}

class CellImpl(elem:int, next:Cell) implements Cell {
  void set(n:int) { elem := n } /*elem:ref(int)*/
  int get() { elem } /*elem:int*/
  Cell getNext() { next } /*next:Cell*/
  Cell createIntBuffer(aux:Cell, size:int) { /*aux:createIntBuffer*/
    if (size == 0) then {
      new CellImpl(0, null)
    }
    else {
      new CellImpl(0, aux.createIntBuffer(aux, size - 1))
    }
  }
}
```

Listagem A.18: Célula para criar um buffer de inteiros

```
interface IntBuffer {
  int auxGet(buf:IntBuffer, temp:Cell, pos:int);
  int get(buf:IntBuffer, pos:int);
  void auxSet(buf:IntBuffer, temp:Cell, pos:int, val:int);
  void set(buf:IntBuffer, pos:int, val:int);
  Cell getHead();

  usage{ (getHead&auxGet!&get!&auxSet&set)* }
}

class IntBufferImpl(head:Cell) implements IntBuffer {

  int get(buf:IntBuffer, pos:int) { /*buf:auxGet;getHead*/
    buf.auxGet(buf, buf.getHead(), pos)
  }

  int auxGet(buf:IntBuffer, temp:Cell, pos:int){ /*temp:get&getNext, buf:auxGet*/
    if (pos == 0) then {
      temp.get()
    }
    else{
      buf.auxGet(buf, temp.getNext(), pos-1)
    }
  }

  void set(buf:IntBuffer, pos:int, val:int){ /*buf:auxSet;getHead*/
    buf.auxSet(buf, buf.getHead(), pos, val)
  }


  void auxSet(buf:IntBuffer, temp:Cell, pos:int, val:int){ /*temp:set&getNext, buf:auxSet*
    if (pos == 0) then {
      temp.set(val)
    }
    else{
      buf.auxSet(buf, temp.getNext(), pos-1, val)
    }
  }

  Cell getHead(){ /*head:Cell*/
    head
  }

}
```

Listagem A.19: Especificação de um buffer de inteiros

```
main {
  let cell = new CellImpl(0,null) in /*cell:createIntBuffer*/
  let head = cell.createIntBuffer(cell,10) in
  let intbuffer = new IntBufferImpl(head) in { /*intbuffer:get;get;set;get;(get|get)*/
    out(intbuffer.get(intbuffer,0));
    out(intbuffer.get(intbuffer, 1));
    intbuffer.set(intbuffer, 2, 35);
    out(intbuffer.get(intbuffer, 2));
    (intbuffer.get(intbuffer, 0)
       |
        intbuffer.get(intbuffer, 2))
  }
}
```

Listagem A.20: Possível utilização de um buffer de inteiros